DISSENY D'UN ADAPTADOR D'IPS PER A NOCS TOLERANTS A FALLES

Memòria del projecte de final de carrera corresponent als estudis d'Enginyeria Superior en Informàtica presentat per Francesc Vila Garcia i dirigit per Eleni Kanellou i Carles Ferrer.

Bellaterra, Setembre de 2009

El firmant, Carles Ferrer , professor del Microelectrònica i Sistemes Electrònics de la Universitat Autònoma de Barcelona

CERTIFICA:

Que la present memòria ha sigut realitzada sota la seva direcció per Francesc Vila Garcia

Bellaterra, Setembre de 2009

Firmat: Carles Ferrer

Índice general

Índice de figuras

Índice de cuadros

Listings

Capítulo 1

Introducción

1.1. Motivación y objetivos

Durante los últimos años, muchas aplicaciones en distintos ámbitos se implementan usando sistemas empotrados. Reproductores MP3, PDAs y teléfonos móviles son claros ejemplos de estas aplicaciones. Estos dispositivos implementan, cada vez más, una gran variedad de aplicaciones y protocolos para comunicarse entre sí, ya sean protocolos de acceso a internet, reproductores de video o videojuegos en 3D; aumentando así tanto su funcionalidad cómo su complejidad.

Para el diseño y posterior implementación de estos dispositivos, tenemos distintas maneras de afrontar este problema. Entre muchas soluciones, tenemos los System-On-Chips (SoCs), los Multi-Procesor System-On-Chips (MPSoCs) (Sistemas SoC con múltiples procesadores) y las Network-On-Chips (NoCs). Nos centramos en estos tres diseños porqué un SoC supone una reducción en el tiempo de desarrollo y un decremento en los costes de producción. Además estas tres metodologías están relacionadas. Un MPSoC, cómo su propio nombre indica; es un SoC con más de un procesador, y un NoC es un sistema MPSoC cambiando la capa de comunicación entre los elementos.

Los sistemas SoC suelen tener un procesador, memoria y varios periféricos

e interfaces con el exterior (ethernet, wifi, USB entre otros), todo conectado alrededor de un bus. Cómo podemos observar en la figura 1.1, tendremos un sistema formado por un microprocesador (el ARM), interfaces para interactuar con el exterior (puertos serie, USB, conversor analógico-digital, bus CAN) y distintos periféricos cómo timers, y reguladores de voltaje, interconectado por el bus AMBA.

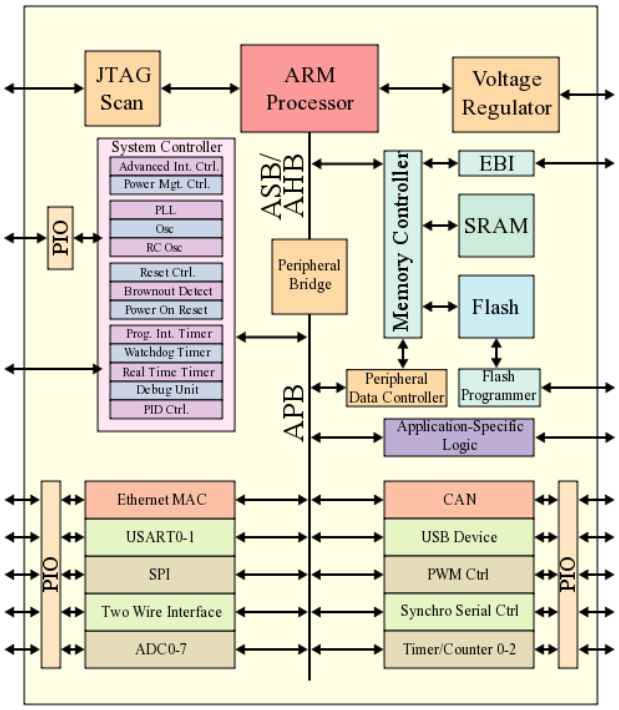

Figura 1.1: Ejemplo de un diagrama de un SoC.

Podemos ver que con estos componentes ya tenemos un sistema completo, que puede cumplir diversas funciones de un modo sencillo. Dependiendo del programa que se cargue en memoria y de los periféricos e IPs que le conectemos, podemos conseguir desde un reproductor de video a un teléfono móvil o un reproductor MP3.

Dada la evolución de la tecnología, cada vez conseguimos una densidad de integración mayor y podemos integrar más componentes en un mismo chip. Al insertar más procesadores, microcontroladores o Digital Signal Procesors (DSPs) al

sistema obtenemos los MPSoC. Siguiendo este diseño, obtendremos las ventajas que nos ofrece un sistema SoC, y además tendremos varias unidades de proceso.

Todos los componentes del MPSoC están interconectados por un bus del sistema compartido, y se deberá controlar el acceso a este. Por lo tanto, para poder hacerlo necesitamos un sistema de arbitraje, dónde decidiremos el componente que accede al bus. Este tipo de acceso puede suponer un cuello de botella, ya que no todos los componentes se podrán conectar con los otros a la vez. Además nos encontramos delante de una situación poco escalable. Otro problema que nos encontramos es que a la vez que se añaden componentes, aumenta la complejidad del sistema de arbitraje, cosa que nos puede penalizar en rendimiento y/o área. Para solucionar estos inconvenientes de los MPSoC, encontramos las NoC, que nos proporcionan un enfoque distinto a la comunicación entre los distintos componentes de un MPSoC.

Las NoC aplican la teoría de comunicación de las redes convencionales a la comunicación dentro de un mismo circuito. Por lo tanto, se cambian las comunicaciones usando un bus compartido y un sistema de arbitraje a una red distribuida formada por enlaces punto a punto. La red está formada por los componentes (nodos) conectados con routers, permitiendo así la comunicación de todos los nodos con el resto.

Aún usando la teoría de redes, al ser una red montada en un mismo circuito, no será necesario implementar todas las características de una red convencional, a la vez que también tendremos que contemplar e implementar casos que no se dan en estas. Por ejemplo, un detalle que diferencia las redes de ordenadores convencionales con las NoCs es que la topología será fija, es decir, que no variará en el tiempo.

Con este nuevo paradigma conseguimos aumentar la escalabilidad del sistema y aumentar la densidad de integración. Pasaremos de buses largos a enlaces punto a punto más cortos. El problema que tienen los buses, es que ocupan mucho espacio, y es muy difícil disminuir la distancia entre las pistas que lo componen, ya que existe riesgo de interferencias. Usando los enlaces de las NoC, conseguiremos

reducir el tamaño de las conexiones teniendo enlaces más cortos y más estrechos, es decir, menos pistas; y nos deshacemos del cuello de botella que supone tener un bus compartido. Podremos explotar el paralelismo en las conexiones entre los componentes porqué los enlaces pueden operar simultáneamente.

Por las razones anteriores, es interesante aplicar el paradigma de las NoC en el diseño de sistemas multiprocesador, ya que ofrece varias ventajas respecto a los sistemas con bus compartido, convirtiéndolo en una opción para el futuro diseño de este tipo de sistemas.

El estudio realizado en este trabajo detalla una manera de conseguir el paso de las arquitecturas MPSoC a arquitecturas NoC, de la forma más directa posible, es decir, que sea posible pasar de un sistema MPSoC existente a una NoC sustituyendo sólo la capa de comunicaciones. Para que esto sea posible, se creará una interfaz que actuará cómo adaptador entre los procesadores y la red.

Esta interfaz será la encargada de convertir entre el protocolo del bus usado para el MPSoC, que será el protocolo que ya utilizan todos los componentes; y el protocolo usado en la red. De este modo, la interfaz actuará cómo un adaptador entre los procesadores y la red. Con esto, conseguiremos que para los distintos procesadores que integran el sistema, el hecho de estar conectados por una red sea transparente. Éstos actuarán cómo si estuvieran conectados a un bus, y será la interfaz la encargada de hacer los cambios necesarios. Así conseguiremos que no sean necesarias modificaciones en los *IP cores* que formen nuestro sistema multiprocesador.

Además de la interfaz necesitamos un componente software. Esta parte será una librería con varias funciones que presentará al programa un modo de acceder a la red, a modo de driver[1] de la interfaz. Con este conjunto de funciones se facilitará la conversión del programa que se ejecuta en los procesadores para poder comunicarse en red. Es importante tener en cuenta que la interfaz sólo nos permite conectar los procesadores a la red sin realizar ningún cambio en el componente

[1]Librería que controla un periférico, en este caso serán un conjunto de funciones para poder comunicarnos usando la interfaz.

hardware del sistema. El software que se ejecuta sí que tiene que estar preparado para conectarse a la red usando nuestro driver.

Cómo veremos más adelante, el trabajo se centrará en hacer una interfaz que sea capaz de convertir del protocolo del bus AMBA al protocolo de la red. Usaremos la GRLIB, que es una librería de IP cores (entre otros contiene el procesador LEON3) para el diseño de sistemas SoC y MPSoC.

Con la interfaz no será necesario ningún cambio en el diseño del LEON3, ya que se comunicará usando el bus AMBA, y sólo se tendrá que usar nuestro driver para poder programar un software capaz de aprovechar la red.

Así pues, tendremos varios bloques formados por el LEON3, una RAM y nuestra interfaz; conectados a través de una serie de *routers* formando una NoC.

1.2. Pasos seguidos y estructura de la memoria

Los pasos que vamos a seguir para conseguir la interfaz son los siguientes:

- Estudiar el estado del arte centrándonos en el trabajo realizado en NoCs y especialmente interfaces.
- Estudiar varios procesadores (con sus entornos)
- Diseñar la interfaz
- Simular la interfaz
- Implementar un driver

Para poder lograr el objetivo de este trabajo, primero hemos hecho un estudio del “estado del arte”. Con dicho estudio pretendemos conseguir una idea de los progresos que se han hecho hasta el momento en el campo de las NoC, con especial atención en el diseño de las interfaces para la red. Además, sumando la

investigación que hemos realizado sobre distintos procesadores, y de los entornos de desarrollo a los que están asociados; nos permitirá acotar y estudiar detenidamente las investigaciones realizadas sobre NoCs con el entorno escogido. Este estudio lo veremos con más detalle en el capítulo 2 de la memoria.

Al mismo tiempo que realizamos el estudio anterior, también estudiaremos los distintos procesadores que tenemos disponibles para la realización del proyecto. Ya que los procesadores suelen tener un entorno asociado para el diseño de SoC y/o MPSoC, también tomaremos en cuenta dicho entorno. Junto al paso anterior, ya que están íntimamente relacionados; obtendremos una base sobre la que construiremos nuestro diseño.

En el capítulo 3 de la memoria, describiremos el entorno del proyecto. Explicaremos con detalles las herramientas y dispositivos usados en la realización de la interfaz.

Una vez descrito el estado del arte y el entorno de trabajo, en el capítulo 4 veremos la planificación del proyecto. Explicaremos los pasos que realizaremos y el tiempo invertido en cada paso.

En el capítulo 5, dedicado al diseño de la interfaz vamos a entrar en detalle sobre el funcionamiento de la misma. Veremos que funcionalidades aporta y cómo se han conseguido, además de ver el porqué se han tomado distintas decisiones de diseño. Una vez explicado esto, introduciremos el driver realizado para la interfaz, empezando por el modo de programar y de acceder a nuestro dispositivo y explicando las funciones que podemos realizar con él.

El último paso dedicado al diseño de la interfaz, explicado en el capítulo 6 de la memoria; contiene las pruebas hechas al diseño. En este capítulo explicaremos los resultados de la simulación de nuestro diseño, y presentaremos programas de test, que pondrán a prueba nuestra interfaz. Con esto conseguiremos validar el funcionamiento del core y del driver diseñados para la interfaz.

Capítulo 2

Estado del arte

Existen varios artículos que exploran la construcción de NoCs. Algunos, cómo Henrique C. Freitas et al. [2]; realizan un estudio del rendimiento de los sistemas NoC frente a sistemas MPSoC, cambiando distintos parámetros, cómo el número de procesadores a conectar o el número de buses del sistema. En él se muestra que las soluciones basadas en NoC son más escalables que un MPSoC, montando un sistema con varios procesadores Microblaze tanto en red (formando la NoC) cómo con el bus OPB (formando un MPSoC). En [5], Li Ping Sun et al. también realizan un estudio sobre el rendimiento de sistemas NoC, pero, al contrario que [2], sigue una arquitectura híbrida, es decir, los procesadores disponen de su propia memoria, y no comparten la misma que está conectada por una red.

Otros estudios, cómo el realizado por Slobodan Lukovic et al. en [3]; describen una metodología para el diseño de NoCs usando también el Microblaze de Xillinx. En este artículo se describen los componentes que forman la red y las Network Interfaces que conectan entre sí los procesadores y los bloques de memoria, aunque la importancia del artículo reside en la modificación del flujo de diseño del EDK[1] de Xillinx para añadir la posibilidad de crear NoCs.

Nos encontramos que hay muchos más artículos sobre distintos diseños de

[1]EDK es el *Embedded Development Kit*. Es un grupo de herramientas de Xillinx para el diseño de sistemas empotrados. También contiene un conjunto de *IP cores.*

NoCs que artículos centrados en la interfaz, aunque sea una parte importante que nos permite poder adaptar las IP existentes a una arquitectura en red. Aún así, nos encontramos con artículos cómo el de M.D. van de Burgwal [6] que se centra en el estudio de una interfaz, Hydra. El uso de esta interfaz es para adaptar a una red un procesador configurable, el Montium TP. Sanjay Pratap Singh, en [4]; describe una interfaz genérica. Lo consigue añadiendo una capa totalmente independiente entre el procesador y dicha interfaz. De este modo consigue que el control sobre la interfaz no cambie, y se pueda adaptar a cualquier *IP core* sólo cambiando la capa más cercana al core.

En [6], Andrei Rădulescu et al., presentan el diseño de una interfaz que desacopla la comunicación de la computación. Es decir, actúa en el la capa de transporte del nivel OSI, que es la primera capa dónde los servicios que proporciona la interfaz son independientes de la implementación de la red. Proponen un diseño modular, con una parte común (el kernel de la NI) que es extensible a través de módulos (shells). Estos módulos le proporcionan funcionalidades extra (cómo por ejemplo, implementación de varios protocolos de bus).

En resumen, los diversos estudios realizados sobre las interfaces de red, proponen un diseño que adapta los IP cores existentes, permitiendo su conexión en red. Las interfaces coinciden en que no se tenga que cambiar el diseño de los componentes a conectar, ocultando la implementación de la red.

Capítulo 3

Descripción del entorno

3.1. GRLIB

La GRLIB es una librería de *IP cores*, especialmente diseñada para el desarrollo y construcción de sistemas SoC.

Un *IP core* es una unidad de código, celda o diseño de chip que es reutilizable. Por lo tanto, una librería de *IP cores* es un conjunto de unidades, con una relación entre sí.

En concreto, la GRLIB se compone de un procesador (el LEON3) y de varios componentes, por ejemplo, controladores para distintos tipos de memoria; comunicados usando un mismo bus, el bus AMBA.

Ya que todos los componentes que la forman están centrados para comunicarse con este bus, nos aseguramos la interoperabilidad entre todos los componentes y su fácil interconexión. Cómo veremos más adelante, el bus nos proporcionará una manera sencilla de añadir nuevos elementos al sistema (ya sean de la misma librería o creados por nosotros mismos) porqué el árbitro asigna automáticamente recursos a los componentes conectados y permite su control desde el procesador. Por lo tanto, podemos considerar que disponemos de un bus *Plug & Play*.

Todos los componentes de la GRLIB están diseñados para poder ser sintetizados en una gran variedad de FPGAs de distintos fabricantes. Esta característica nos quita una restricción importante sobre el hardware que tenemos que usar. La FPGA que usaremos será la VirtexII-Pro de Xillinx integrada en un kit de diseño de Digilent. En concreto es el kit Digilent XUP. Usar el kit nos da flexibilidad porqué integra, además de la FPGA; varios conectores, posibilidad de conectar memoria externa y conectores de expansión para conectarlo a distintas placas, convirtiéndolo en una solución ideal de prototipaje. Este kit tiene un precio especial para instituciones académicas, reduciendo el coste total de los materiales del proyecto.

Por último, el flujo de diseño en este entorno es intuitivo, ya que proporciona una interfaz única para la simulación y posterior síntesis del proyecto. Esta interfaz se compone de una serie de scripts y *Makefiles* que se gestionan a través de una interfaz gráfica que nos permitirá configurar los componentes que queremos del sistema. Esta interfaz es independiente de la FPGA que se utilice. La GRLIB contiene varios diseños de ejemplo. Son sistemas multiprocesador parametrizables y se pueden cambiar el número de procesadores, periféricos y memoria con los mencionados scripts cómo veremos más adelante. Hemos modificado este sistema por uno con un solo procesador y la interfaz de red.

3.1.1. LEON3

El procesador utilizado para la elaboración de este proyecto es el LEON3. Este procesador nos lo proporciona la GRLIB en forma de *IP core*, por lo tanto, cumple con las características de todos los componentes de la librería. Está desarrollado por GAISLER, la misma compañía que distribuye la GRLIB.

El LEON es una implementación de la versión V8 de la arquitectura Sparc. Fue diseñada por Sun Microsystems en 1986. El Sparc V8 es un procesador puramente *big endian*, e implementa un conjunto de instrucciones Reduced Instruction Set Computer (RISC).

El LEON3, es un procesador con licencia GPL, por lo tanto de código abierto. Tiene dos versiones, el LEON3 que es gratis y el LEON3FT, que no lo es. La diferencia es que el LEON3FT tiene varios cambios para ser un procesador tolerante a fallos, cómo códigos de corrección de errores en registros y cache que no tienen penalización de tiempo respecto al LEON3.

El LEON3FT, por su tolerancia a fallos, es un procesador que se usa en aplicaciones aeroespaciales. La versión 2 de este procesador fue diseñada por encargo por la agencia espacial europea. La versión no tolerante a fallos, se usa, por ejemplo; en aplicaciones multimedia (unas 70 compañías lo usaban en el 2008).

Este procesador tiene un gran rendimiento cómo se puede ver en la Tabla 5 de [1], dónde se hace una comparativa de los tiempos de proceso para un mismo programa de distintos procesadores, quedando en segundo lugar el LEON3 (y en primer lugar el LEON2). El procesador es totalmente configurable con los scripts de la GRLIB.

3.1.2. AMBA

El AMBA es el estándar de facto para las comunicaciones entre los distintos componentes de un SoC. Por esto facilita una metodología de reutilización de los distintos componentes (nos ofrece un estándar y compatibilidad con distintos componentes de varios fabricantes). En la GRLIB es el bus que conecta todos los IP cores que la componen, por ejemplo, el LEON, la memoria y nuestra interfaz.

La especificación está pensada para facilitar la interconexión de varios procesadores, convirtiéndolo en un candidato perfecto para el diseño de sistemas SoC o MPSoC. En el proyecto usamos la versión 2 de la especificación, que dispone de dos protocolos que cubren las necesidades, tanto de los componentes que requieren una alta tasa de transferencia cómo los que no requieren tanto ancho de banda. Los protocolos (no los protocolos, los buses) están conectados por un adaptador (*bridge*) Podemos ver en la figura 3.1, que los dos buses se pueden comunicar a través del bridge. Los protocolos del bus son:

- El Advanced High-performance Bus (AHB): Es la parte del bus dónde se conectan los componentes del sistema que necesitan una tasa de transferencia más alta. En este bus conectaremos el procesador, la memoria y la interfaz de red.

- El Advanced Peripherial Bus (APB): Es dónde se conectan los periféricos que no requieren tanto ancho de banda, cómo controladores de teclado o puertos serie. En nuestro diseño conectaremos la DSU (Debugging Support Unit) que nos proporciona acceso a la E/S estándar desde el programa y un *timer* para el *polling*.

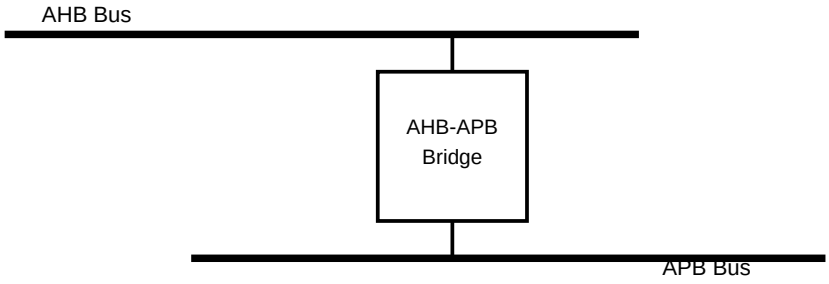

Figura 3.1: Los dos tipos de bus AMBA conectados por el bridge.

3.1.3. Otros componentes

Para completar el entorno de diseño, los distribuidores de la GRLIB, además de la librería nos ofrecen varias herramientas más:

BCC: Un compilador cruzado para facilitar el desarrollo de programas para el sistema. Con él vamos a poder programar en un lenguaje de más alto nivel que el ensamblador los distintos programas de test para la interfaz y la red.

TSIM: Un simulador del procesador que nos permitirá probar el código de un modo más rápido que sintetizando el diseño en una FPGA o cargando el programa en el Modelsim (el simulador de VHDL). Cómo inconveniente es

que no podemos añadir nuestra interfaz (sólo emula el LEON3) pero nos servirá para ver el comportamiento de nuestro programa.

GRMON: Un monitor de *debug*, que se comunica con la DSU de nuestro sistema y nos permite el *debugging* del sistema ya sintetizado en la FPGA. Con esta herramienta podremos observar el comportamiento más realista del sistema, ya que se ejecuta sobre la FPGA. Gracias al GRMON, se podrá obtener una funcionalidad cómo con el TSIM, pero con nuestra interfaz ya añadida al sistema.

ENTORNO GRÁFICO: La GRLIB nos proporciona un entorno gráfico de configuración para los distintos componentes. Se puede acceder a él de dos modos distintos: *make xgrlib* y make xconfig. Desde la ventana del primer comando podemos lanzar los comandos de simulación, síntesis y place and route. También podemos escoger el programa que se lanzará para cada tarea y lanzar la ventana de configuración. El segundo comando nos lanza la configuración del diseño. Desde ésta podremos escoger los componentes que queremos que tenga nuestro sistema y configurar el procesador para adaptarlo a nuestras necesidades. Podemos ver las interfaces en la figura 3.2 y 3.3.

Para empezar un nuevo diseño con la GRLIB, después de extraerla, haremos una copia de uno de los diseños que vienen con la librería. Luego, ya podremos empezar a incluir nuestros ficheros en el sistema, añadiendo nuestro código y editando el top del sistema (y el Makefile para añadir nuestro dispositivo). Podemos ver la estructura básica de directorios de la GRLIB y la creación de un proyecto nuevo llamado NI en la figura 3.4

3.1.4. Red

La red para la que está pensada la interfaz unirá varios procesadores conectados mediante *routers*. No se conectarán sólo los procesadores, si no que mediante

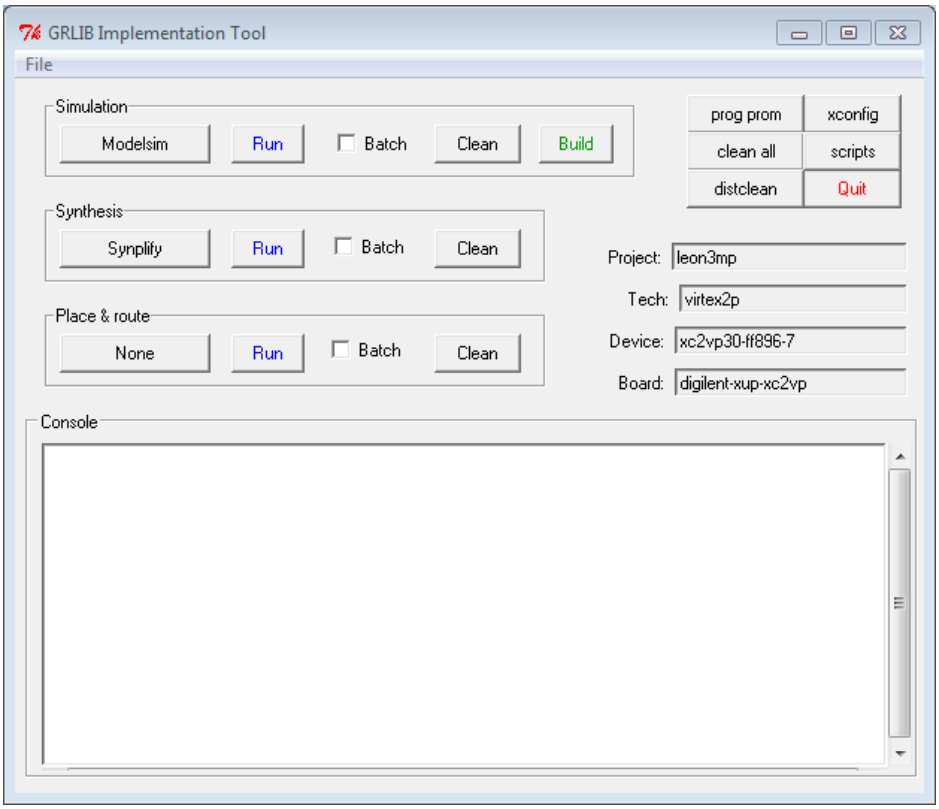

Figura 3.2: Pantalla de *make xgrlib*. Vemos los botones de acceso a las herramientas de simulación, síntesis, y *place and route*.

la interfaz se unirán unos bloques compuestos por procesador, memoria y nuestra interfaz (entre otros componentes) cómo se puede ver en la figura 3.5.

La función de los componentes se describe a continuación (en orden de derecha a izquierda y de arriba a abajo):

LEON3: Es el procesador del sistema. Se encargará de ejecutar el programa que corresponda al bloque.

Memoria: Es dónde se guardará el programa que tiene que ejecutar el bloque.

Interfaz: Es la encargada de adaptar las comunicaciones entre el procesador (bus AMBA) y la red. Es el objetivo de este trabajo.

Árbitro AMBA: Es el componente que se encargará de controlar el acceso al

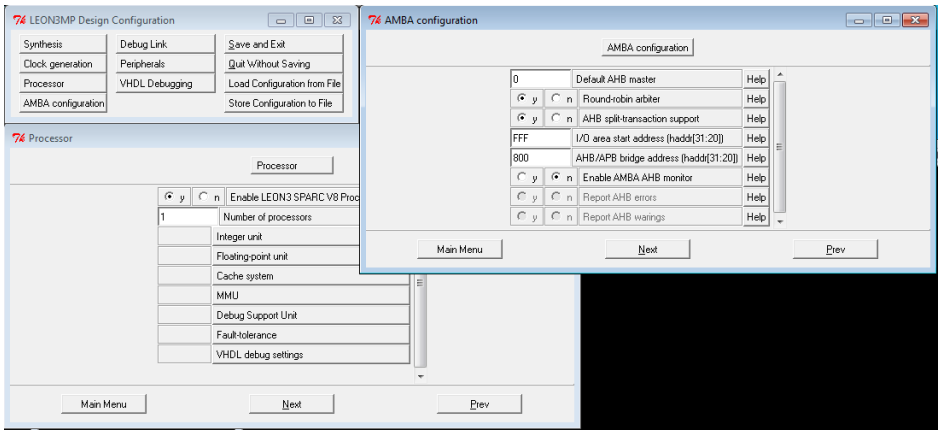

Figura 3.3: Pantallas de *make xconfig*. Podemos ver la pantalla de configuración (arriba a la izquierda) y las opciones del procesador y del bus AMBA.

bus para que todos los componentes se puedan comunicar entre sí. Nos lo proporciona la GRLIB.

AHB-APB bridge: Cómo hemos visto anteriormente, el bridge se encargará de comunicar los dos buses (el AHB y el APB). También nos lo proporciona la GRLIB.

Timer: Con éste podremos fijar un intervalo de polling para comprobar si ya hemos recibido algún paquete. Forma parte de la GRLIB.

Como vemos, cada bloque contiene un procesador y una memoria, por lo tanto, la red tendrá una topología híbrida, es decir; los procesadores se conectan a las memorias usando un bus (como un SoC) pero los bloques se conectarán entre sí por una red (ver la figura 3.6)

3.1.5. Protocolo

La red usará una técnica de *switching* llamada *wormhole switching*. En esta técnica, los paquetes se dividen en varios trozos más pequeños llamados *flits*[1] para su transmisión por la red. Tendremos entonces, tres tipos de *flits*:

[1]Flit es un acrónimo de FLow control biTS.

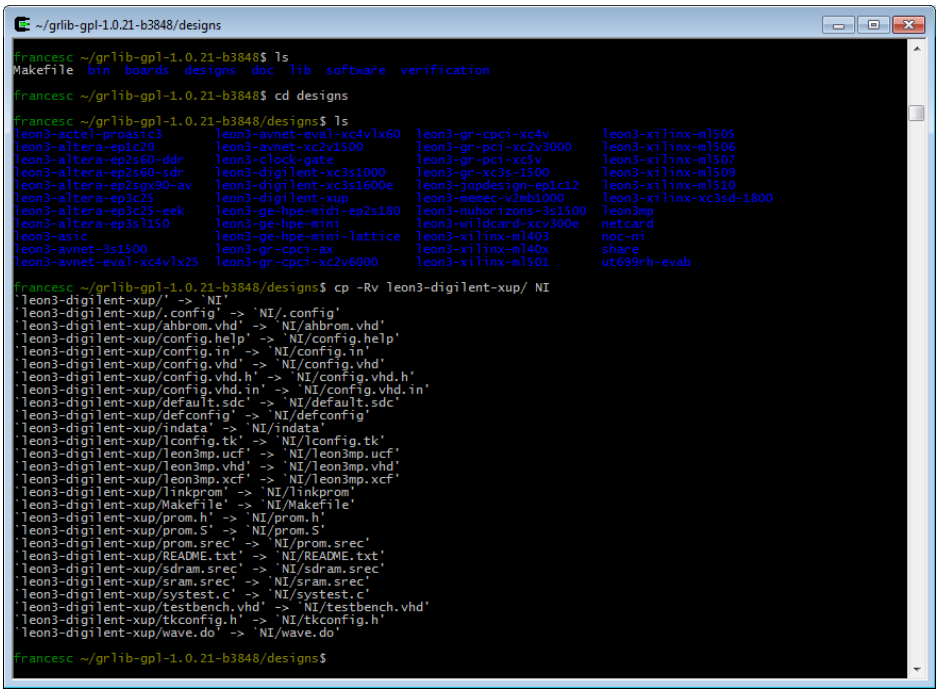

Figura 3.4: Vemos la estructura de directorios de la GRLIB y la copia del diseño base para la placa Digilent XUP para crear el proyecto NI.

- El *header flit* es el primer *flit* que se transmite, y contiene información sobre la ruta que debe seguir el paquete. Es decir, contiene la dirección de destino de éste. Su función es reservar el camino por dónde pasarán el resto de *flits*.

- Los *body flit* son los siguientes *flits* que se envían después del *header flit*. Pueden ser un número variable dependiendo del tamaño del paquete a transmitir. En nuestro caso es un número fijo, ya que todos los paquetes tienen el mismo tamaño.

- El *tail flit* es el *último flit* que se manda, y su función es la de ir liberando los recursos reservados por el *header flit*. De este modo, el camino seguido quedará libre otra vez

Así nos encontramos que un mismo paquete estará repartido por varios switches al largo de su recorrido por la red. De hecho, de aquí viene el nombre de

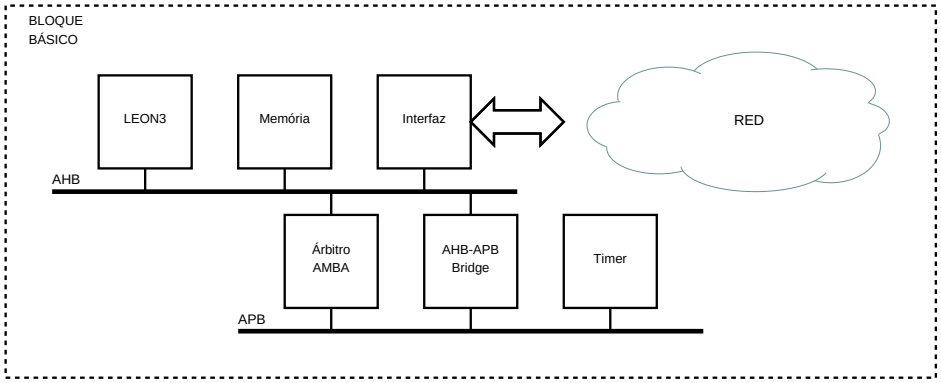

Figura 3.5: Estructura de un bloque básico. El elemento que será unido por los *routers*.

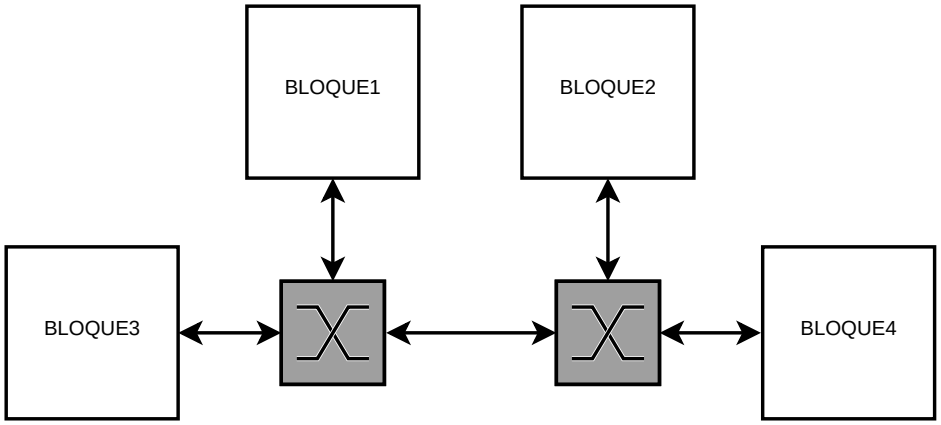

Figura 3.6: Topología ejemplo de una red. Los bloques están descritos en la figura 3.5.

esta técnica, ya que al estar repartido de este modo crea una imagen parecida a un gusano (*worm*).

Capítulo 4

Planificación

Podemos ver en la figura 4.1, un diagrama de GANTT con la planificación del proyecto.

Hemos dividido el proyecto en las siguientes tareas:

1. Estudio de procesadores: Dedicaremos un tiempo estudiando distintos procesadores para usarlos cómo elemento para conectar con nuestra interfaz. Dedicaremos unos 10 días a esta tarea.

2. Estudio del estado del arte: En esta tarea dedicaremos otros 20 días en estudiar artículos que se han hecho sobre interfaces de red para NoCs.

3. Familiarización GRLIB: Dedicaremos un par de semanas en familiarizarnos en el entorno de diseño del proyecto. En nuestro caso es la GRLIB.

4. Desarrollo de la interfaz: Será la parte del trabajo que nos lleve más tiempo. La podremos dividir en:

    a) Envío de paquetes: Dedicaremos la mitad del tiempo de diseño en realizar la parte de envío de la interfaz.

    b) Recepción de paquetes: Dedicaremos la otra mitad del tiempo en diseñar la recepción de paquetes de la interfaz.

   c) Driver: Durante el diseño de la interfaz, dedicaremos tiempo en el diseño del controlador de la interfaz.

5. Test: Dedicaremos un mes y medio en realizar los tests al diseño de la interfaz.

6. Memoria: Dedicaremos un mes en la redacción de la memoria.

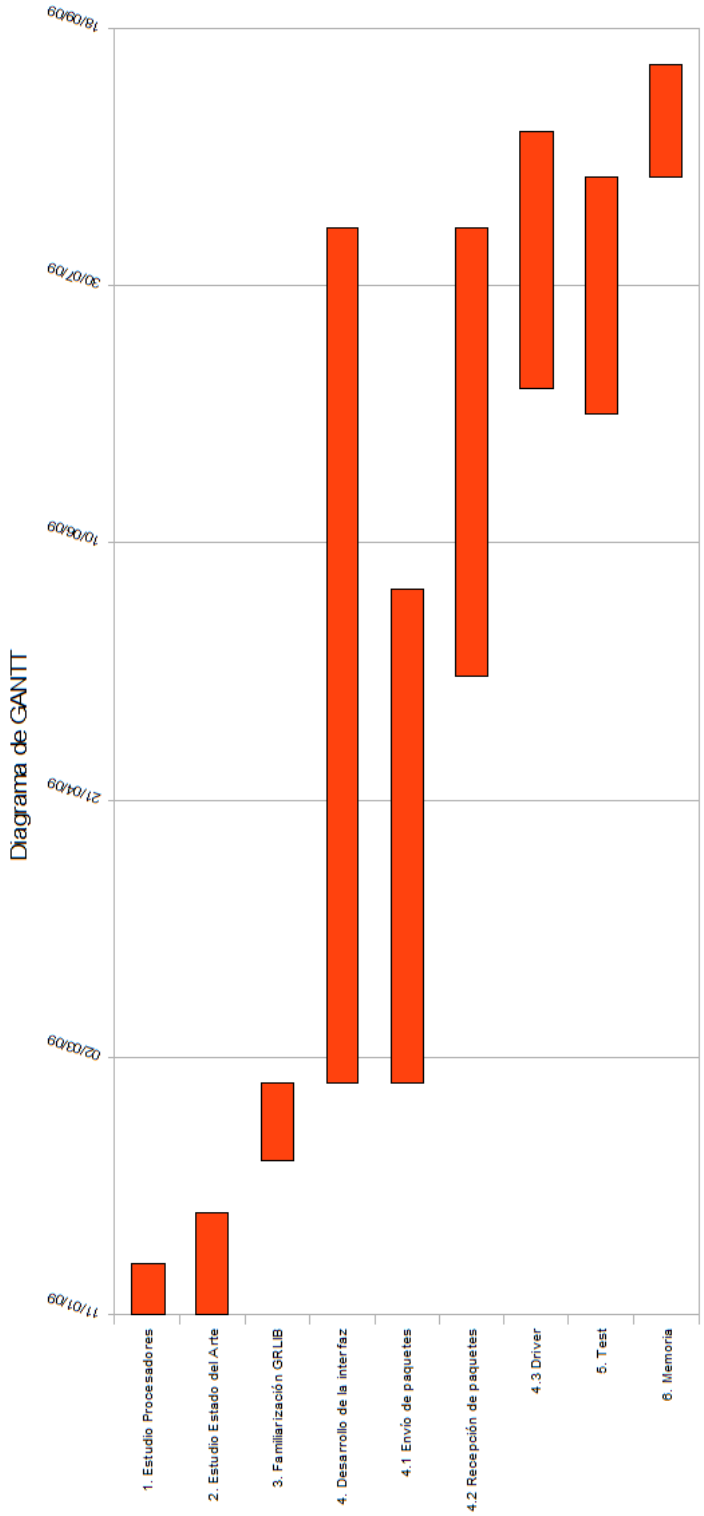

Figura 4.1: Diagrama de GANTT con la planificación prevista del proyecto.

Capítulo 5

Interfaz de red

En este capítulo discutiremos el diseño de la interfaz de red. Esta interfaz servirá de adaptador entre el procesador y la red. De este modo conseguiremos que para el procesador el acceso a la red sea transparente ya que la interfaz ocultará los detalles.

5.1. Descripción del funcionamiento

Para que la interfaz pueda ocultar los detalles de la red al procesador, esta tendrá dos partes diferenciadas. Una parte que actuará cómo un periférico más del bus Advanced Microcontroller Bus Architecture (AMBA) y que será la encargada de comunicarse con el LEON, y otra parte encargada de comunicarse con el *switch* al que está conectada. Entonces, la interfaz hará una conversión del protocolo del bus al de la red, y viceversa.

Antes de empezar a explicar cómo funciona la interfaz, vamos a explicar cómo funciona el bus AMBA.

La interfaz está conectada a la parte Advanced High-performance Bus (AHB) del bus, y el acceso se realiza accediendo a un mapa de memoria. Cada dispositivo conectado al bus mapea una cantidad de memoria, y entonces, cada vez que

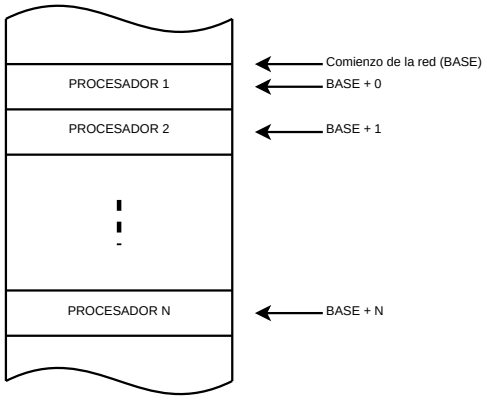

Figura 5.1: Ejemplo de mapa de memoria para una red de *N* nodos.

desde el procesador accedemos a una de esas posiciones, nos comunicamos con la interfaz directamente. Al acceder a una posición podremos leer o escribir en ella. Por lo tanto, si la interfaz tiene *mapeados* desde la dirección `0x70000000` a la `0x70010000` y escribimos en la dirección `0x70000FFF`, el árbitro del bus se encargará de activar la interfaz y transmitirle el dato que hemos escrito. Para hacer esto, activará la señal de selección de la interfaz (*hsel*), la señal de escritura (*hwrite*) y dejará la dirección y los datos en *haddr* y *hwdata* respectivamente.

Entonces, y teniendo en cuenta que la red es fija y no varía al largo del tiempo, podremos mapear todos los procesadores conectados en memoria. Para comunicarnos, sólo tendremos que acceder a la posición de memoria que corresponda. Podemos ver un mapa de memoria de ejemplo en la figura 5.1.

Cómo se puede observar en el mapa de memória, los procesadores están mapeados desde la dirección *BASE* a la dirección *BASE + N*. Teniendo en cuenta que la dirección *BASE* será conocida por el programa, las direcciónes de las interfaces serán $0, 1, 2, ..., N$.

Este sistema tampoco consumirá mucha memoria, ya que cada procesador conectado sólo necesita una palabra de memoria (4 bytes). Vemos que si, por ejemplo, queremos crear una red de 100 procesadores, mapearíamos sólo 400 bytes (no llegamos a 1 KByte, y la memoria puede ser del órden de MBytes).

Una vez visto el sistema de comunicación del procesador con la interfaz, pa-

saremos a ver la descripción de las dos operaciones básicas que podremos realizar con la interfaz: enviar y recibir datos.

5.1.1. Envío de datos

Nuestra interfaz, siguiendo con las operaciones del bus AMBA; podrá enviar datos en dos ocasiones: cuando quiera enviar un dato a otro procesador, o cuando quiera escribirlo.

En los dos casos, el procedimiento es el mismo, aunque cambian las opciones del paquete que se envía.

Para enviar un dato se deberán seguir dos pasos:

1. Escribir en la dirección mapeada del dispositivo con el cual nos queremos comunicar las opciones necesarias (lectura/escritura o si el paquete es una ráfaga o es único).

2. Escribir en la misma dirección de memória el dato a enviar (en el caso que estemos enviando algo) o 0 en cualquier otro caso.

Cómo resultado la interfaz enviará un paquete al procesador correspondiente con el dato que enviemos y la indicación de escritura o un paquete vacío con la opción de lectura. Dependerá del otro procesador el procesar el paquete y realizar las operaciones necesarias.

5.1.2. Recepción de datos

La interfaz dispone de un búfer de recepción de datos para cada nodo de la red. Cuando la interfaz recibe datos, los coloca en el búfer correspondiente dependiendo del origen. Esto es posible, porqué la red es estática, y en todo momento sabemos el número de procesadores que están conectados.

El procesador, entonces, realiza un *polling* cada cierto tiempo (dictado por el timer que tenemos incorporado en el sistema) para comprobar si hay datos nuevos disponibles.

En el caso que no haya datos disponibles, no se hace nada, pero si hay algún dato disponible, el procesador lo leerá y hará lo que sea necesario para tratarlo.

Para hacer el *polling*, el procesador leerá de la posición de memória del nodo del cual quiera comprobar si hay algun paquete disponible. En caso de que esté disponible (se indicará con un bit de control en la palabra que se lea), lo podrá obtener leyendo otra vez de esa misma posición.

5.1.3. Resumen

Al largo de este capítulo veremos con más detalle los procesos de escritura y de lectura. Cómo resumen, veremos que existe una correspondencia directa entre las dos operaciones que podemos realizar en el bus AMBA (lectura o activar *hread* y escritura o activar *hwrite*) con los dos procesos que hemos descrito anteriormente. Esta correspondencia se muestra en la tabla 5.1

Señal AMBA	**Operación**
hread	Recepción de datos de la red.
hwrite	Envío de datos. Puede ser tanto una escritura (envío de un dato a otro procesador), cómo una lectura (solicitud de un dato a otro procesador).

Cuadro 5.1: Resumen de las operaciones de la interfaz según los valores de señales del bus AMBA.

5.2. Descripción de los módulos principales

En esta sección se van a describir con un poco más de detalle la función de los módulos principales de nuestra interfaz. En la figura 5.2 podremos ver un esquema con todos los módulos de la interfaz y cómo están conectados.

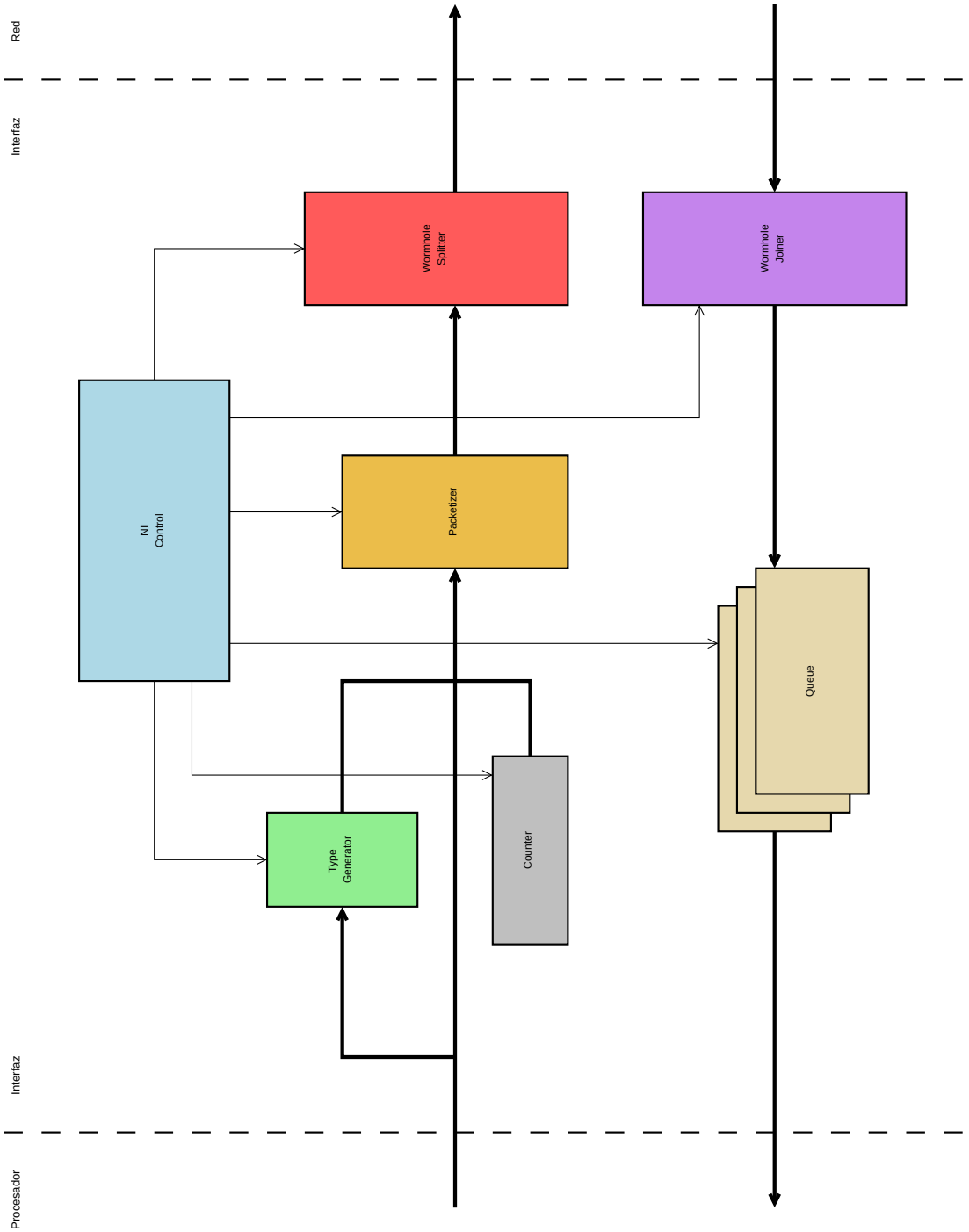

Figura 5.2: Diagrama con los módulos de la NI. La línea gruesa es el camino de datos, y la fina las señales de control.

5.2.1. Unidad de control

La unidad de control es la que realiza tódo el trabajo para sincronizar los componentes y adaptarse tanto al protocolo del bus AMBA cómo al protocolo de la red. En la figura 5.3 podemos ver un diagrama de estados del comportamiento de esta unidad.

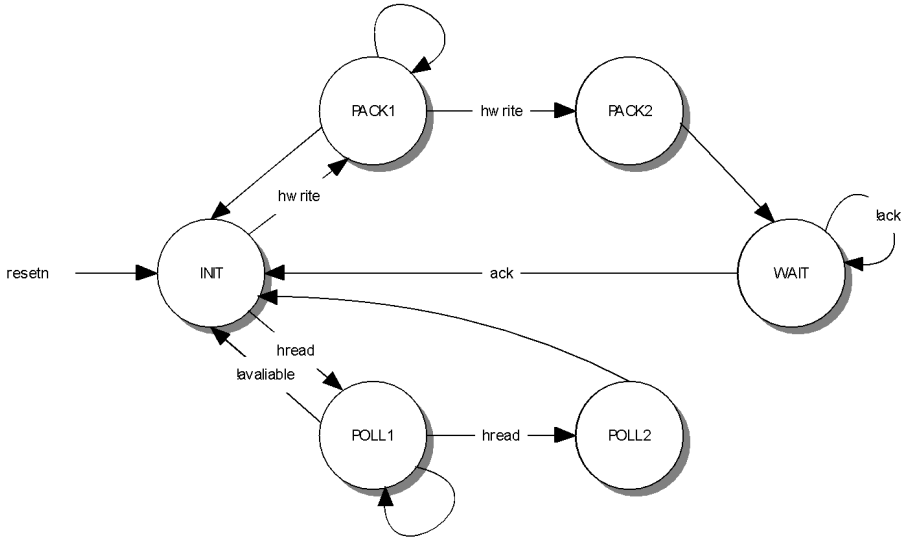

Figura 5.3: Diagrama de estados del funcionamiento de la unidad de control.

Este diagrama está simplificado, pero se puede observar que está partido. Desde el nodo INIT (el nodo inicial en el cual espera que lleguen las peticiones) salen dos caminos. El camino de la parte de arriba de la figura (PACK1 - PACK2 - WAIT) es el que se sigue cuando se mandan los paquetes, y el camino de abajo (POLL1 - POLL2) es el que se sigue cuando se reciben.

Comportamiento del envío

Cómo hemos visto, un envío se corresponde a una o dos escrituras en la dirección de memória correspondiente al procesador con el cual nos queremos comunicar.

Primero de todo se escriben las opciones del envío, y después, en la misma dirección de memória se escriben los datos (si es necesario). El formato de la palabra para las opciones tendrá el formato descrito en le figura 5.4.

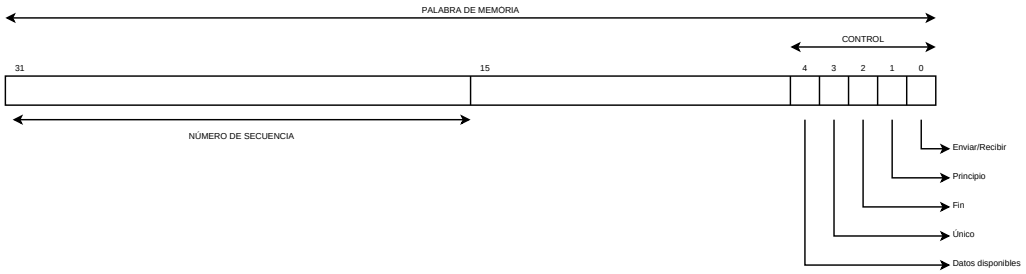

Figura 5.4: Formato de las opciones en la comunicación con la interfaz

El significado de los últimos bits es el siguiente:

Enviar/Recibir indica si el paquete que se tiene que enviar es una solucitud de lectura o de escritura al procesador correspondiente. Si tiene el valor 0, será de lectura, y en otro caso, de escritura.

Principio indica si es el principio de una ráfaga de varios paquetes.

Fin indica si es el fin de una ráfaga.

Único indica que el paquete es único.

Datos disponibles es un campo que se usa en la recepción de paquetes. Se discutirá en el siguiente apartado.

Número de secuencia contiene el número de secuencia del paquete.

No podremos tener activos a la vez los bits de *principio*, *fin* o *único*.

Una vez hayamos escrito esta palabra de control, en el caso de que sea una transferencia de escritura, es decir, que querramos enviar un dato, tendremos que escribir el dato a enviar en la misma dirección de memoria.

Se ha escogido este sistema para evitar que el procesador se quede bloqueado en el caso que se desee recibir un dato y este no esté disponible. Una vez hecha la

petición de un dato a otro nodo de la red, el procesador podrá seguir ejecutando su programa. Ya recibirá el paquete cuando se ejecute la rutina de *polling*.

Comportamiento de la recepción

Gracias al *timer* que hemos incorporado a nuestro sistema, podremos ejecutar una rutina de *polling* a la interfaz, para comprobar si tenemos algún paquete que espera en la cola de recepción.

Para poder comprobar si existen paquetes en la cola, lo haremos leyendo de la posición de memória. Entonces, la interfaz nos retornará la palabra de control (ver figura 5.4. Si el bit número cuatro está activo, tendremos un paquete disponible para la lectura. En este caso, podremos leerlo de esa misma posición de memória.

Al igual que con el envío de datos, se ha decidido este sistema para que el procesador no se quede bloqueado esperando datos que aún no han llegado.

5.2.2. Buffer de recepción

Estos *buffers* almacenan los paquetes que van llegando de la red para que el procesador los vaya cogiendo durante el proceso de *polling*.

Es una cola circular, dónde los paquetes se ponen en el órden del que llegan.

Al guardar los paquetes, lo hacemos de la manera más equitativa posible. Ya que tenemos una cola para cada procesador de la red, no nos encontraremos en el caso que un procesador envíe muchos paquetes y no tengamos lugar para ellos.

5.2.3. Packetizer

Este módulo es el encargado de crear los paquetes para su posterior envío por la red. Podemos observar el formato de paquete en la figura 5.5.

Podemos ver una descripción de los campos a continuación:

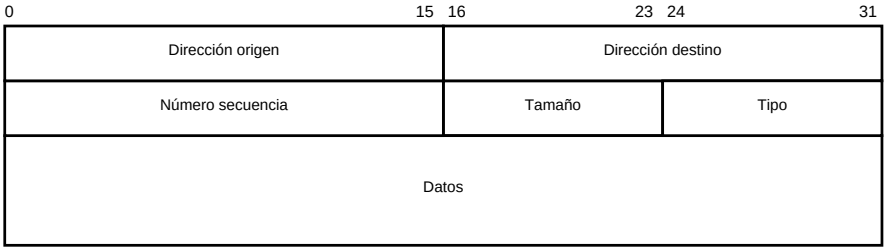

Figura 5.5: Formato del paquete que se envía por la red.

Dirección de origen contendrá la dirección de origen del paquete. Esta dirección será fija para cada interfaz de la red. Esto es así porqué la red será estática.

Dirección de destino contendrá la información sobre el destino del paquete. Se obtendrá del campo *haddr* del bus AMBA. No se envia el campo directamente, ya que no tenemos porqué tener mapeada la interfaz en el mismo lugar en todos los procesadores. Por lo tanto, se va a enviar el *offset* desde la dirección base (ver el mapa de memória en la figura 5.1).

Número de secuencia contendrá el número de paquete para poder ordenar los paquetes en caso de que se envíe una ráfaga. El número se obtiene de un contador que se incrementa para cada paquete que se envía.

Tamaño contiene el tamaño de la sección de datos del paquete. En el caso de una solicitud de lectura, tendrá el valor 0.

Tipo contiene el tipo del paquete. En este campo indicaremos si es una solicitud de lectura de un dato o una de escritura.

Datos contendrá los datos del paquete que se envía.

5.2.4. Wormhole-splitter

Cómo hemos comentado en el capítulo anterior, la red usa una técnica de *wormhole switching*. Con este módulo, dividimos cada paquete creado en diversos *flits*.

Este módulo ya se encarga de poner las etiquetas necesarias a los *flits*. El primero llevará los flags de inicio, el último llevará el flag de último y el resto no llevaran ningún flag.

5.2.5. Wormhole-joiner

Es el módulo contrario al wormhole splitter. Cuando llegan *flits* de la red este módulo se encarga de ensamblarlos.

Por las características de la red, podemos suponer que cuando llega un *flit*, estos van a llegar en órden. También podemos suponer que no nos van a llegar *flits* correspondientes a otros paquetes.

5.3. Diseño del driver

En este apartado veremos el diseño del controlador de la interfaz de red. Se compondrá de un conjunto de funciones que facilitarán el acceso a la red desde los programas que carguemos en el procesador.

5.3.1. Objetivos del driver

El driver será el encargado de controlar la interfaz desde un programa para el procesador LEON. Por lo tanto, tendrá las siguientes funciones:

- Inicializar la interfaz, para que esta pueda enviar y recibir paquetes.
- Proporcionar una función para enviar datos al resto de procesadores.
- Instalar la rutina de *polling* para comprobar las colas de recepción de la interfaz.

- Proporcionar una función para poder recuperar los paquetes que han llegado.

5.3.2. Descripción de las funciones

Enviar paquete

La función que envía un paquete tiene el siguiente prototipo:

```
int enviar_paquete (int destino, int flags, void data)
```

- El destino es el identificador del procesador al que queremos enviar los datos.
- Los flags nos indican el tipo de paquete a enviar. Para aplicar varios tendremos que hacer una OR entre ellos. Las distintas opciones son las siguientes:
  - SOL_LECTURA para enviar una solicitud de lectura al otro procesador.
  - SOL_ESCRITURA para enviar una solicitud de escritura al otro procesador.
  - PKT_UNICO indica que sólo se enviará un paquete.
  - PKT_INICIO_RAFAGA indica que vamos a empezar a enviar un conjunto de paquetes.
  - PKT_FIN_RAFAGA indica que hemos acabado de enviar la ráfaga de paquetes.
- data contiene los datos a enviar. Tiene un tamaño fijo de 4 bytes.

Es importante notar que no podremos tener activos a la vez los flags de lectura y de escritura.

Las opciones de ráfaga se usan para la ordenación de los paquetes en el destino. Si el paquete es único se envía directamente, pero si es parte de una ráfaga, al

leer retornamos también el número de secuencia, para que el programa los pueda reordenar.

Recibir paquete

La función de recibir un paquete tendrá el siguiente prototipo:

```
int recibir_paquete (int destino, int* n_seq, void* data)
```

El parámetro destino cumple con la misma función que el de la función de enviar paquete. El resto de parámetros son:

- n_seq es una variable que la función actualiza con el número de secuencia del paquete.
- data es la variable que después de llamar a la función contiene los datos del paquete. Esta función será la que se llamará en la rutina de polling con las distintas direcciones de los procesadores de la red.

Capítulo 6

Simulación del diseño

En este capítulo veremos las pruebas realizadas sobre la interfaz para comprobar que el diseño es correcto. Se han realizado pruebas funcionales simulando las unidades que forman el sistema. De este modo conseguiremos validar el diseño de la interfaz.

6.1. Pruebas funcionales

Estas pruebas son las que demuestran que los componentes funcionan correctamente. Vamos a mostrar la simulación de los componentes:

6.1.1. Unidad de control

Hemos dividido las pruebas del diseño en dos partes: el envío y la recepción de paquetes.

Podemos ver el comportamiento del envío en la figura 6.1.

Primero vemos la parte marcada en rojo. Es lo que provoca la transición al estado de *PACK1*. Cuando se activa, también se activan *hready* y *enable_register*,

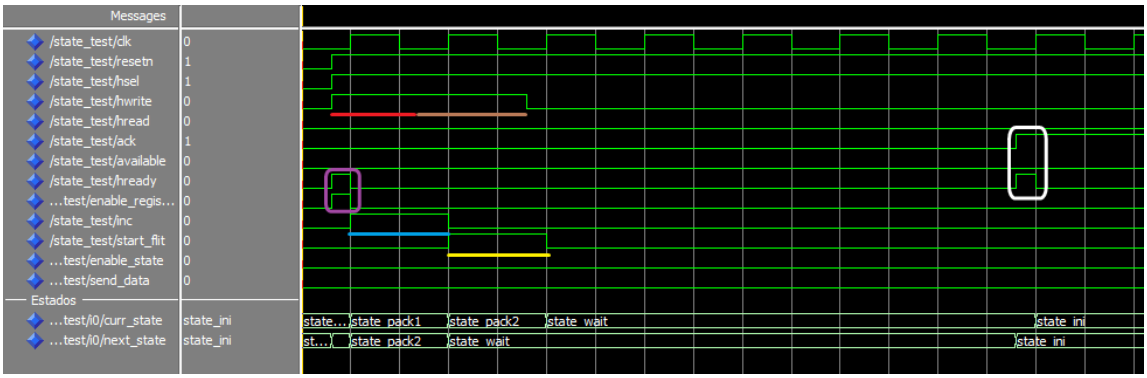

Figura 6.1: *Waveform* de la unidad de control durante el envío.

que activará un registro para guardar la palabra de control. Entonces, también incrementamos el número de secuencia activando la señal *inc* (marcado en azul).

Al recibir el segundo hwrite, significa que estamos enviando los datos, por lo tanto activamos la señal *start_flit* que nos empezará a dividir el paquete para mandarlo por la red.

Cuando hemos acabado de partir el paquete, se activará la señal *ack* y volveremos al estado inicial, preparados para enviar o recibir más paquetes.

En la figura 6.2 podemos ver el comportamiento de la unidad de control cuando se lee un paquete de la cola.

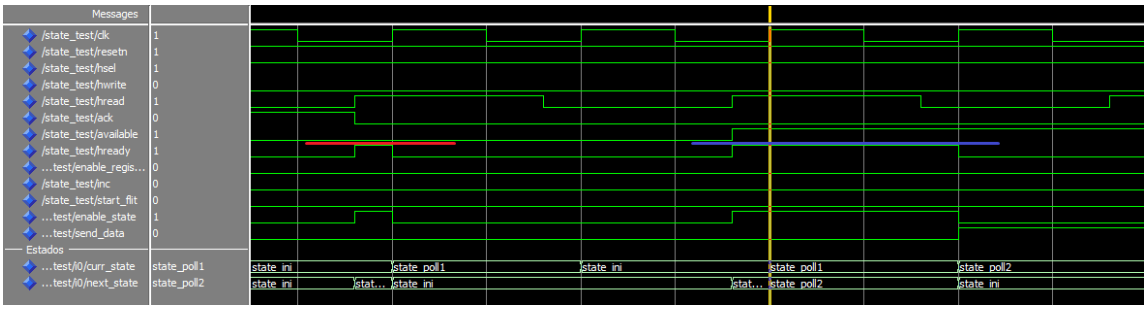

Figura 6.2: *Waveform* de la unidad de control durante la recepción.

Si nos fijamos en la parte señalada en rojo, cuando intentamos leer y *available* vale 0, volvemos al estado inicial (ya que no hay ningún paquete disponible). Al poner *available* a 1, si hacemos un *hread* para leer el paquete, pasamos por los

dos estados (POLL1 y POLL2) dónde enviamos los datos al procesador. Luego volvemos al estado inicial.

6.1.2. Buffer de recepción

Podemos ver el comportamiento del buffer de recepción en la figura 6.3.

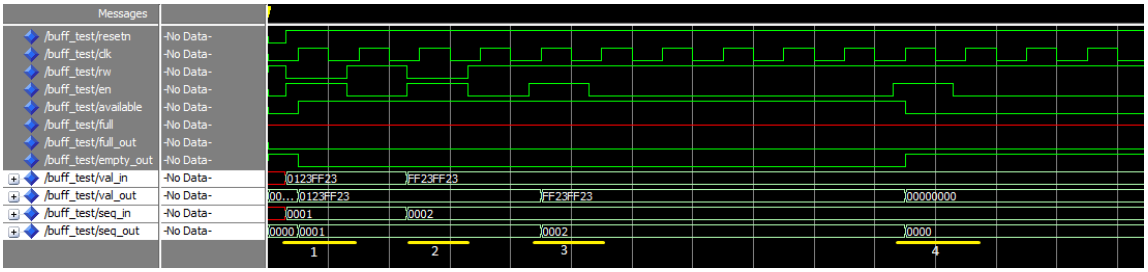

Figura 6.3: *Waveform* del buffer de recepción.

Hemos realizado las siguientes acciones:

1. Insertar un paquete con datos *0x0123FF23* y número de secuencia *0x0001*
2. Insertar un paquete con datos *0xFF23FF23* y número de secuencia *0x0002*. En la salida del módulo aún tenemos el primer paquete, ya que es una estructura *FIFO*.
3. Al leer el paquete, eliminamos el primero y sólo nos queda *0xFF23FF23*.
4. Aquí leemos el último paquete. Vemos que available está a 0 y nos indica que no tenemos más paquetes disponibles en esta cola.

6.1.3. Packetizer

Para comprobar este módulo, vamos a comprobar la creación de un paquete con las características mostradas en la tabla 6.1.

Según la figura 5.5 el paquete resultante deberá ser:

Dirección origen	1 (0000000000000001)
Dirección destino	3 (0000000000000011)
Número de secuencia	10 (0000000001010)
Tamaño	1 (00000001)
Tipo	Escritura + Único (00001100)
Datos	01010100100111001111011010011101

Cuadro 6.1: Tabla con los valores del paquete. Para el formato, ver la figura 5.5.

0000 0000 0000 0001 0000 0000 0000 0011

0000 0000 0000 1010 0000 0001 0000 1100

0101 0100 1001 1100 1111 0110 1001 1101

Que en hexadecimal es:

$0x10003000A010C549CF69D$

Comprobamos en la figura 6.4 la salida de nuestro módulo.

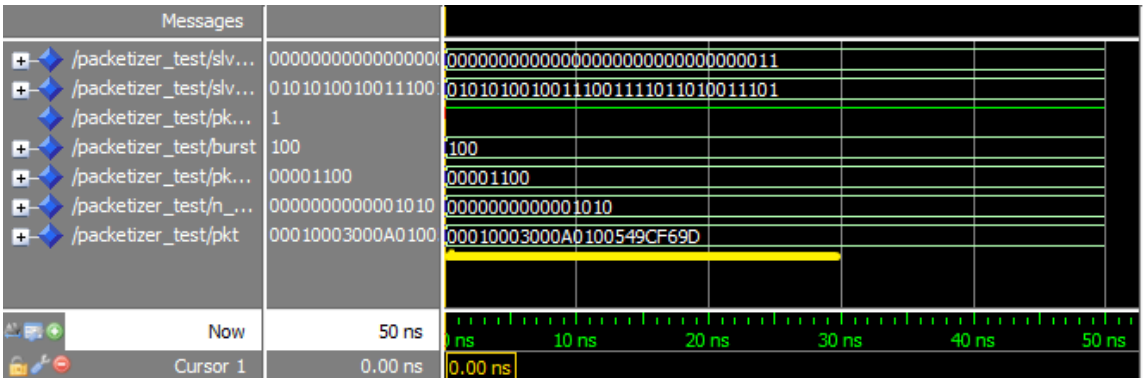

Figura 6.4: *Waveform* del packetizer.

La parte señalada en amarillo nos muestra el paquete una vez ya ensamblado.

6.1.4. Wormhole splitter

Si cogemos el paquete anterior y lo dividimos en *flits* veremos el comportamiento del módulo *wormhole-splitter*. Lo podemos ver en la figura 6.5.

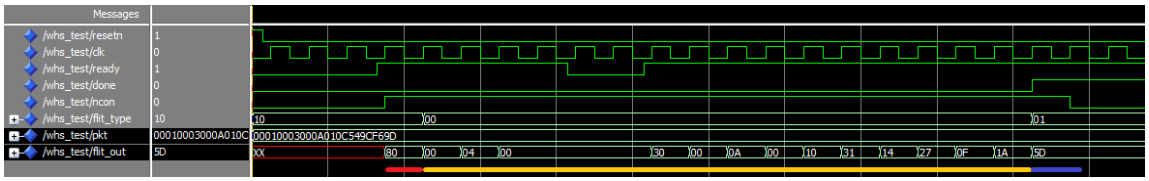

Figura 6.5: *Waveform* del módulo *wormhole splitter*.

La parte marcada en rojo es el *start flit*, la parte marcada en amarillo son los diversos *middle flits* y la parte en azul es el *end flit*. Podemos ver, también, que cuando envía los *flits*, si la señal *ready* que proviene del router al que estará conectado está a 0, es decir, que no está preparado para recibir *flits*, el módulo se espera hasta que esté disponible, y que cuando acaba, avisa que ya se ha mandado (señal *done*) y un ciclo después desactiva la señal *ncon*, encargada de pedir una conexión al router dónde está conectada la interfaz.

6.1.5. Wormhole joiner

Para el testeo de este módulo, se ha procedido a conectarlo justo después del *wormhole splitter*.

De este modo, si generamos los *flits* de un paquete y luego los unimos con el módulo, el paquete que deberemos obtener será el mismo.

Para comprobar el funcionamiento, hemos creado un *testbench* automatizado, que envía el paquete, lo divide en flits, y luego reconstruye el paquete. Una vez hecho esto, lo comparamos con el paquete original para ver si es igual. La parte relevante del código del *testbench* lo encontramos en el listado 6.1.

Listing 6.1: Código del *testbench* automatizado

```
1 wait for 600 ns;
2 assert pkt1 = packet_out
3 report "Test failed"
4 severity failure;
5 assert pkt1 /= packet_out
```

```
6 report "Test succeeded"
7 severity failure;
```

Al ejecutar el código, comprobamos si los paquetes son iguales. Si no lo son, imprimimos "Test failed". Luego para acabar el programa, si los dos paquetes son iguales, imprimimos "Test succeeded".

Podemos ver en la figura 6.6 que los dos paquetes son iguales:

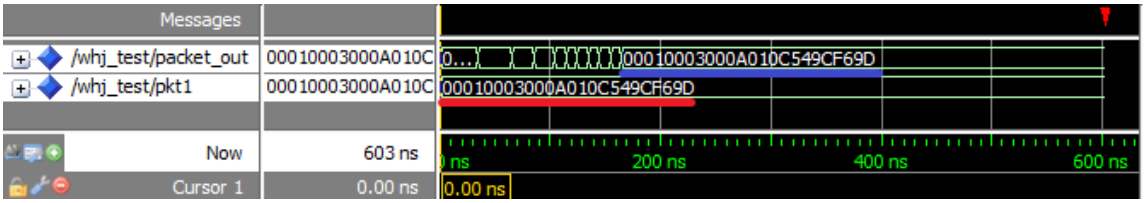

Figura 6.6: *Waveform* del módulo *wormhole joiner*.

El paquete marcado en rojo es el paquete que mandamos, y el marcado con azul es el paquete reconstruido.

Capítulo 7

Conclusiones

En este capítulo expondremos las conclusiones a las que hemos llegado después de la realización de este proyecto.

En este proyecto hemos estudiado distintos enfoques a la implantación de una interfaz para NoCs.

En nuestro caso, hemos diseñado una interfaz simple, capaz de adaptar un procesador (el LEON3) al protocolo usado en una NoC. Se ha diseñado la interfaz para intentar que sea lo más transparente posible, de este modo, el procesador se comunica con otro del mismo modo que si estuviera usando el bus AMBA, sólo cambia el código del programa ejecutado en el procesador.

Durante la implementación, hemos descartado opciones cómo métodos que aseguran fiabilidad punto a punto, dejando esta tarea al programador de las aplicaciones para el LEON. Así, la interfaz sólo actúa de adaptador entre el bus y la NoC.

A lo largo de la construcción de la interfaz, hemos visto varias mejoras aplicables al diseño, aunque añadiríamos complejidad a esta. Por ejemplo, podríamos añadir fiabilidad en las comunicaciones punto a punto, para liberar al programa de esta tarea. También podríamos incluir la posibilidad de enviar más datos en un mismo paquete, aprovechando más el ancho de banda de la red y aumentando

el throughput. Otra mejora más que podríamos incluir es cambiar el proceso de polling por interrupciones. Cada vez que llegue un paquete, en lugar de esperar a que se compruebe si hay alguno disponible, se lanzaría una interrupción para que el procesador la procese.

Para concluir, es importante resaltar la importancia del estudio de las interfaces de red, ya que es una parte fundamental del sistema. Si podemos conseguir que las comunicaciones sean independientes del diseño de los módulos que forman la red (procesadores por ejemplo), conseguiremos adaptar los IP cores existentes al paradigma de las NoC. Al hacer que las interfaces oculten los detalles de implementación de la red, conseguimos desacoplar el diseño de los IP cores y de la red, con lo que los podremos diseñar al mismo tiempo, reduciendo el tiempo necesario para construir un sistema.

Capítulo 8

Lista de acrónimos

SoC System-On-Chip

MPSoC Multi-Procesor System-On-Chip

NoC Network-On-Chip

DSP Digital Signal Procesor

RISC Reduced Instruction Set Computer

AMBA Advanced Microcontroller Bus Architecture

AHB Advanced High-performance Bus

APB Advanced Peripherial Bus

Bibliografía

[1] Sergio De Florio, Eberhard Gill, Simone D'Amico, and Andreas Grillenberger. Performance comparison of microprocessors for space-based navigation applications.

[2] H. C. Freitas, D. M. Colombo, F. L. Kastensmidt, and P. O. A. Navaux. Evaluating network-on-chip for homogeneous embedded multiprocessors in fpgas, 2007. ID: 1.

[3] Slobodan Lukovic and Leandro Fiorin. An automated design flow for noc-based mpsocs on fpga. *Rapid System Prototyping, IEEE International Workshop on*, 2008.

[4] Sanjay Pratap Singh, Shilpa Bhoj, Dheera Balasubramanian, Tanvi Nagda, Dinesh Bhatia, and Poras Balsara. *Generic Network Interfaces for Plug and Play NoC Based Architecture*, volume 3985/2006 of *Reconfigurable Computing: Architectures and Applications*, pages 287–298. Springer Berlin / Heidelberg, 2006.

[5] Li Ping Sun, El Mostapha Aboulhamid, and J. P David. Network on chip using a reconfigurable platform, 2003. ID: 1.

[6] M.D. van de Burgwal, G.J.M. Smit, G.K. Rauwerda, and P.M. Heysters. Hydra: an energy-efficient and reconfigurable network interface.

Firmat: Francesc Vila Garcia

Bellaterra, Setembre de 2009

Resum

L'aparició d'un nou paradigma per al disseny de sistemes multiprocessador, les NoC; requereixen una manera d'adaptar els IP cores ja existents i permetre la seva connexió en xarxa. Aquest projecte presenta un disseny d'una interfície que aconsegueix adaptar un IP core existent, el LEON3; del protocol del bus AMBA al protocol de la xarxa. D'aquesta manera i basant-nos en idees d'interfícies discutides en l'estat de l'art, aconseguim desacoblar el processador del disseny i topologia de la xarxa.

Resumen

La aparición de un nuevo paradigma para el diseño de sistemas multiprocesador, las NoC; requieren un modo para adaptar los IP cores existentes y permitir su conexión en red. En este proyecto se presenta el diseño de una interfaz que consigue adaptar un IP core existente, el LEON3; del protocolo del bus AMBA al protocolo de la red. De este modo, y basándonos en algunas ideas de interfaces discutidas en el estado del arte, conseguimos desacoplar el procesador del diseño y topología de la red.

Abstract

The emergence of a new paradigm for the multiprocessor systems design, the NoC; requires a way to adapt existing IP cores to this new communication backbone, in order to be able to connect them to a network. In this project, we present an interface design that achieves to adapt an existing IP core, the LEON3 processor; from the AMBA bus to the network protocol. In this way and taking into account some ideas from the designs discussed in the state of art, we can decouple the processor design from the network design and topology.