Implementación de una red en chip en un procesador RISC-V

NoC implementation of a RISC-V processor

Por

David Davó Laviña

Trabajo de Fin de Grado del Grado en Ingeniería Informática

Facultad de Informática
Universidad Complutense de Madrid

Óscar Garnica Alcázar & Juan Lanchares Dávila

Madrid, 2021-2022

Agradecimientos

En primer lugar quiero agradecer a mi novia, Leidy, por estar conmigo durante más de 7 años queriéndome, apoyándome y ayudándome y, sobre todo, por recordarme que de vez en cuando hay que descansar un poquito y despejarse.

También quiero dar las gracias a mi familia; en particular a mis padres, Anselmo y Gloria, por aguantarme mientras duraba el proyecto y desde siempre, y a mi tía, Toñi, por los consejos de estilo y ayuda con la memoria.

Este trabajo tampoco habría sido posible sin mis tutores del TFG, Óscar y Juan, que han estado semana por semana siguiéndome y aportándome ayuda cuando más atascado me encontraba. También a muchos de mis antiguos profesores, sobre todo a aquellos *de letras* que confiaron en mí cuando nadie más lo hizo y me hicieron cambiar.

A la gente de la universidad que no puede estarse quieta: FDIst, OTEA y ASCII que, a pesar del poco tiempo libre con el que cuentan sus profesores y estudiantes, deciden sacar un poco para dedicarlo a hacer cosas chulas (y, a veces, dudosamente legales).

También a los compañeros y amigos que hice por el camino. Aunque ya no nos veamos tanto sé que seguís ahí y merecéis mi gratitud: Zero, Leila y Victor, Pont, Ela y Null, Pascal, Marina, Daster, Yago y muchos más... siempre me acordaré de todos vosotros. Igual a mis amigos del pueblo: Mario, Sandra, Paco, Rubén, Aritz, Hugo, Carlos... sé que cuando empecé a estudiar cada vez más, y más lejos, fuimos perdiendo el contacto, pero tampoco olvidaré esos años de juegos online nocturnos en los que comenzaron mis andaduras de sysadmin.

No sin olvidar a los chicos y chicas de MEL, iniciativa del pueblo con la que crecí y me eduqué y con la que se me presentaron oportunidades increíbles. En especial a mis educadores, Esther y Jorge, que supieron ver en mí algo que el resto no quiso. Sin ellos, desde luego no sería quien soy hoy.

Finalmente, me gustaría agradecer a todos aquellos gigantes sobre los que se apoya nuestro trabajo, a todos los hackers, piratas y magos que se quedan hasta tarde, a los que dieron el alma a una nueva máquina, la máquina de los sueños, y crearon imperios, catedrales y bazares. Gracias a todos ellos que, a lo largo de décadas de historia han permitido que este trabajo, y muchos otros, sean posibles.

Resumen

Las redes en chip (NoC) son una tecnología emergente alternativa a los métodos de interconexión convencionales en la que se aplican los métodos de diseño de redes a las conexiones entre módulos. Permiten una mayor escalabilidad, abstracción, flexibilidad y resiliencia que las conexiones intra-chip convencionales.

Aunque las NoC han sido ampliamente usadas para la intra-comunicación entre procesadores y dispositivos complejos, en este Trabajo de Fin de Grado aplicamos esta metodología dentro de la unidad de ejecución de un procesador basado en la arquitectura libre RISC-V: el SWerv-EL2.

Por otro lado, en las FPGAs la reconfiguración dinámica posibilita añadir, sustituir y eliminar elementos heterogéneos en tiempo de ejecución, permitiéndonos incorporar funcionalidades conforme van siendo necesarias, o aportar redundancia y reemplazar módulos defectuosos para hacer nuestro diseño más tolerante a fallos.

Para ello, hemos estudiado a fondo y diseñado una NoC, tratando de minimizar los recursos consumidos y el impacto en área de la misma en el procesador. Posteriormente modificamos la unidad de ejecución del procesador para incluir dicha red como interconexión entre los módulos de dicha unidad. En ambos casos usaremos el lenguaje de descripción de hardware SystemVerilog.

Finalmente se comentan los problemas encontrados durante el proyecto, los resultados y conclusiones, y el trabajo futuro posible para la continuación de este proyecto.

Palabras clave

RISC-V, Red en Chip, NoC, Sistemas en Chip, Arquitectura de Computadores, Reconfiguración Parcial Dinámica, FPGA, SystemVerilog

Abstract

Network on a Chip (NoC) is an emergent technology alternative to traditional interconnection methods, in which network design methods are applied to the connexions between modules. They allow greater scalability, abstraction, flexibility and resilience than conventional on-chip networks.

Although NoCs have been widely used to connect processors and complex devices, in this Final Degree Project, we apply this technology inside the execution unit of the RISC-V based processor SWerv-EL2.

Secondly, using a dynamically partially reconfigurable FPGA permits adding, moving, replacing and removing heterogeneous elements at run-time, enabling us to insert features as needed or to provide redundancy by replacing defective modules to make our design fault-tolerant.

For this, we have studied in-depth and designed a NoC, trying to minimize the resources utilized and its impact on the core. Afterwards, we modified the execution unit of the core to include said network, using it to interconnect its submodules. In both cases, we used the hardware description language SystemVerilog.

In the end, we discuss the problems encountered during the project, the results and conclusions, and possible future works.

Keywords

RISC-V, Network on Chip, NoC, System on Chip, Computer Architecture, Dynamic Partial Reconfiguration, FPGA, SystemVerilog

Índice general

Índice de figuras

A menos que se especifique lo contrario, las figuras de este documento son de autoría propia y siguen la licencia de este documento.

Índice de tablas

Parte I.

Introducción

Capítulo 1.

Motivaciones

Una red en chip (NoC) es una arquitectura de comunicación entre submódulos de un circuito integrado. Es una tecnología emergente para los sistemas en chip (SoC), en la que se aprovechan los métodos de redes convencionales para crear conexiones más flexibles entre los distintos dispositivos o módulos del SoC. Además, su diseño modular permite aumentar la productividad en las fases de diseño, reutilizando los elementos generados (topología, encaminadores, interfaces de red...) para conectar módulos heterogéneos.

Gracias a su escalabilidad y eficiencia (tanto energética como de coste de desarrollo y área), su uso ha aumentado de manera notable recientemente, sobre todo para conectar los distintos *núcleos* en las nuevas arquitecturas *multicore* y *manycore*.

Por último, también se está usando para incorporar técnicas de detección y corrección de errores y mejorar la calidad de servicio, haciendo la comunicación mucho más resiliente y tolerante a fallos.

Cuando una partícula ionizante impacta con un dispositivo electrónico, puede producirse un cambio de estado en sus señales o *soft error*. Debido al reducido tamaño de los transistores, los circuitos integrados son cada vez más susceptibles a fallos, y su tasa de fallos aumenta cuando se expone directamente a radiación, como es en aplicaciones médicas, militares y aeroespaciales. Estos errores transitorios son detectables y corregibles recargando el estado del circuito. Sin embargo, si la partícula es altamente ionizante, puede dañar permanentemente una parte del circuito.

Por otro lado, usando matrices de puertas lógicas programables (FPGA), podemos realizar reconfiguraciones dinámicas parciales para cambiar en tiempo de ejecución parte del diseño. Al realizar una reconfiguración, nos podemos encontrar que la conexión punto a punto entre dos módulos deje de existir, por lo que es necesario proveer de algún mecanismo de interconexión que permita introducir nuevos elementos en tiempo de ejecución. Por lo tanto, podemos utilizar la NoC para crear un sistema tolerante a fallos para conectar los distintos módulos del diseño. En el caso de que el sustrato sobre el

que se implementa un módulo falle, utilizando la reconfiguración dinámica de la FPGA se puede reimplementar y conectar a otro puerto de entrada a la NoC. Este mecanismo también nos permitiría añadir y eliminar IPs de un sistema según se precise su uso, minimizando el área consumida por nuestro diseño.

Para estudiar la viabilidad de la NoC como mecanismo para aumentar la tolerancia a fallos de un sistema complejo, se va a llevar a cabo una prueba de concepto utilizando el procesador libre SweRV-EL2 de Western Digital, de arquitectura libre RISC-V. En concreto, utilizamos la NoC como red de interconexión de los distintos módulos de la unidad de ejecución de dicho procesador.

El empleo de un procesador abierto como es RISC-V encaja muy bien con la modularidad. Debido a su licencia abierta nos permite modificar implementaciones ya existentes para crear una prueba de concepto. Además, su conjunto de instrucciones, al ser también modular y abierto, nos permitiría añadir un conjunto de instrucciones de control de la NoC de ser necesario.

Chapter 1.

Motivations

A Network on Chip (NoC) is a communication architecture used between the modules of an integrated circuit (IC). It is an emergent technology for Systems on Chip (SoC), in which conventional networking methods are leveraged to create more flexible connections between the different modules of the SoC. Furthermore, Its modular design improves productivity in the design phases, reusing generated elements (topology, routers, network interfaces...) to connect heterogeneous modules.

Thanks to its scalability and efficiency (both in power, development costs and resources), its use has increased significantly recently, especially to connect the multiple cores in new many core and multicore architectures.

Lastly, NoCs are also being used to include error detection and correction methods, and to implement traffic prioritization methods, making the communication more fault-tolerant.

If an ionizing particle hits an electronic device, a change of state or Single Event Upset can be produced. Due to the small size of transistors, ICs are increasingly susceptible to these errors, and its fault rate increases when directly exposed to radiation, for example, in medical, aerospace or military applications. These transient errors can be detected, and corrected by reloading the state of the circuit. However, total ionizing doses can permanently damage part of the circuit.

On the other hand, using field-programmable gate arrays (FPGAs), we can perform dynamic partial reconfigurations (DPR) to change part of the design at runtime. After a DPR, we can find that the point-to-point connection between two modules ceases to exist, so it is necessary to provide some interconnection mechanism that allows new elements to be introduced at runtime. Hence, we can use a NoC to create a fault-tolerant system to connect the modules of the design. In the case that the part of the FPGA in which a module is implemented fails, the module can be reimplemented on free area using DPR, connecting it to another input port of the NoC. This mechanism also allows

adding and removing IPs from a system as they are needed, reducing the overall area consumed by our design.

To study the feasibility of using a NoC as a method to turn a complex system fault-tolerant, a proof of concept is going to be performed using Western Digital's SweRV-EL2, which uses the free and open RISC-V architecture. Precisely, we will use the NoC as an interconnection network for the modules of the execution unit of said processor.

The use of an open processor such as RISC-V fits very well with the modular design. Because of its open licence, we can modify an existing implementation to create our proof of concept. Moreover, its Instruction Set Architecture, being also modular and open, would allow us to add NoC control instructions if needed.

Capítulo 2.

Objetivos

Los objetivos principales de este proyecto son diseñar una NoC de bajo consumo, escalable y parametrizable; y usarla para conectar por lo menos dos elementos dentro de un procesador RISC-V. Cumplir dichos objetivos requerirá:

- Explorar las distintas implementaciones RISC-V y seleccionar una de ellas.
- Familiarizarse con el diseño RTL del procesador seleccionado y elegir los módulos que usarán la NoC.
- Describir los requisitos y características de la NoC a implementar.
- Diseñar, describir en un lenguaje de descripción de hardware, simular la NoC para verificar su correcto funcionamiento y sintetizar la descripción.
- Diseñar y describir los elementos necesarios para permitir conectar la NoC con los módulos del RISC-V.
- Integrar la NoC en el RTL del RISC-V.
- Simular el procesador al completo con programas de prueba para asegurar el correcto funcionamiento del procesador tras la incorporación de la NoC como medio de interconexión.
- Sintetizar los diseños para comprobar la viabilidad del código.

Chapter 2.

Objectives

The main objectives of this project are to design a low-power, low-resources, scalable and parametrizable NoC, and to use it to connect no less than two elements inside a RISC-V processor. Completing said goals will require:

- To explore multiple RISC-V implementations and to choose one of them.
- Familiarizing with the RTL design of the chosen processor, and selecting which modules will use the NoC.
- Describing the requirements and features of the implemented NoC.
- To design, write in a hardware description language, and simulate the NoC to validate its behaviour and synthetize said description.
- Integrating the NoC inside RISC-V's RTL.
- To simulate the modified processor with test programs to ensure the correct functioning of the processor after including the created NoC.
- Synthesize the designs to check the feasibility of the code created.

Parte II.

Estado del arte

Capítulo 3.

RISC-V

> «Computers make excellent and efficient servants, but I have no wish to serve under them.»
> – Leonard Nimoy como Mr. Spock en *Start Trek TOS*

RISC-V es tan solo un ISA (*Instrucion Set Architecture*), es decir, la parte del procesador que debe tener en cuenta el compilador o el programador para generar software ejecutable por una máquina de esa arquitectura [13]. La principal diferencia con otras ISAs como MIPS, ARM o x86 es que RISC-V es una arquitectura de estándar abierto y gratuito, por lo que cualquiera puede diseñar su propio procesador que la implemente.

La arquitectura RISC-V tiene un conjunto de instrucciones *reducido* o RISC (*Reduced Instruction Set Computer*), por lo que cuenta con menos instrucciones que una arquitectura compleja o CISC (*Complex Instruction Set Computer*). Al contrario que en CISC, las instrucciones en RISC son más simples y cada una de ellas realiza una sola función muy concreta. Asimismo, decimos que RISC-V es una arquitectura *load/store*, también llamada de *registro a registro*, en la que el procesador cuenta con un *banco de registros* en el que se realizan las operaciones, y nunca directamente sobre la memoria. Es decir, para ejecutar un cálculo usando datos de la memoria primero es necesario usar instrucciones de *load* para llevar los operandos a los registros, luego realizar el cálculo con todas las instrucciones aritmético-lógicas necesarias y, finalmente, usar un *store* para desplazar el resultado de nuevo a la memoria.

La arquitectura RISC-V es modular, existiendo distintos conjuntos base, y extensiones que pueden añadir funcionalidades a un procesador. Para que un procesador soporte la arquitectura RISC-V no privilegiada debe implementar el conjunto base **I** de 32, 64 o 128 bits; aunque puede usarse el conjunto base **E** de 32 bits para sistemas embebidos, que cuenta con la mitad de registros accesibles [22]. Asimismo, se definen múltiples extensiones que pueden soportarse en cada implementación. A continuación se listan las extensiones no privilegiadas ratificadas en la versión 20191213 del ISA:

M Multiplicación y división entera
A Instrucciones atómicas

F Coma flotante de precisión sencilla (IEEE-754)
D Coma flotante de precisión doble (IEEE-754)
Q Coma flotante de precisión cuádruple
C Instrucciones comprimidas (como el conjunto *Thumb* de ARM)
Zicsr CSR: Registro de Control y Estado
Zifencei Barrera o *fence* para la fase *fetch* de instrucciones

También cuenta con extensiones aún no ratificadas para manipulación de bits, SIMD (*Single Instruction Multiple Data*), operaciones vectoriales, criptografía, JIT (aceleración de lenguajes intepretados), virtualización... Parte del espacio de instrucciones está reservado para la implementación de conjuntos de instrucciones propietarios.

Para identificar las características de la arquitectura de un procesador, suele usarse como nomenclatura el prefijo *RV*, seguido del número de bits y los subconjuntos de instrucciones que soporta (comenzando con la ISA base). Por ejemplo, un procesador de propósito general de 64 bits con multiplicación y división de enteros, instrucciones atómicas y coma flotante sería un procesador RV64IMAFD; mientras que un procesador embebido con la extensión criptográfica sería un RV32EK.

En ocasiones nos referiremos a la extensión **G**, que es una abreviatura de **IMAFDZicsr-Zifencei** y comprende todos los conjuntos necesarios para implementar un procesador de propósito general. Por lo tanto, un procesador RV32/64G es un objetivo estable para el desarrollo de software, pues todas sus extensiones están ratificadas.

La codificación de las instrucciones es fija, y podemos distinguir los siguientes seis formatos en el conjunto base [14]:

R Instrucciones en las que se especifican 3 registros: 2 que almacenan los operandos fuente y un destino en el que guardar el resultado. Para poder realizar más operaciones, este tipo de instrucciones cuentan con bits extra para seleccionar la operación a realizar.
I Instrucciones que incluyen un operando *inmediato* (fijo en la codificación de la instrucción), además de otro registro fuente y un destino.
S Instrucciones de *store*, encargadas de guardar los datos de un registro a la memoria.
SB Instrucciones de ramificación *branch*, que implementan saltos condicionales.
U Instrucciones en las que se especifica un registro destino y un inmediato (de 20 bits, más largo que en las de tipo I).
UJ Instrucciones de salto incondicional.

3.1. Elección del núcleo RISC-V

Para seleccionar el núcleo (*core*) a modificar, se ha explorado la lista de *cores* de RISC-V Exchange [19]. El objetivo principal ha sido buscar un proyecto en el que fuese sencillo embeber una NoC como prueba de concepto, sin importar mucho las características técnicas del procesador, su potencia o sus resultados en *benchmarks*. Por lo tanto, hemos priorizado que su lenguaje de programación fuese conocido, tuviese licencias abiertas, y una amplia documentación y herramientas de desarrollo.

Se ha filtrado la selección buscando Cores escritos en lenguajes conocidos (VHDL o SystemVerilog) y con licencia libre reconocida por OSI (Apache, BSD, GPL o MIT) [18]. Finalmente, la selección se redujo a los siguientes núcleos:

SweRV Core EH1 [RV32IMCZ]: Es superescalar con 2 vías de lanzamiento (dual issue) y segmentación en 9 fases. Ha sido fabricado en tecnología de 28nm, pero cuenta con optimizaciones para FPGA. Creado por Western Digital. Tiene 4 unidades aritmético-lógicas (ALU), 2 *early* y 2 *late*, cada una conectada a una de las vías del ILP [23].

SweRV Core EH2 [RV32IMACZ]: Basado en el EH1, se le añade multithreading para 2 hilos, y por lo tanto también se le agregan instrucciones atómicas [24].

SweRV Core EL2 [RV32IMC]: Es un núcleo mucho más sencillo que el EH1 creado para reemplazar a las máquinas de estados y otras funciones lógicas implementadas en Sistema en Chip (haciendo que puedan ser actualizadas). La segmentación es en solo 4 etapas y es escalar [25].

biRISC-V [RV32IMZicsr]: Superescalar con dual issue y segmentación en 6-7 etapas, tiene un divisor hardware, 1 unidad de load-store y 2 ALUs. Implementa también instrucciones privilegiadas, por lo que puede ejecutar sistemas operativos de propósito general [21].

PicoRV32 [RV32IMC]: Es un pequeño procesador de menos de 2000 LUTs, por lo que caben múltiples en una FPGA.

Algunos otros núcleos y herramientas que se consideraron y fueron descartados en fases tempranas (principalmente por el lenguaje de programación o su complejidad) fueron los siguientes:

WARP-V Está escrito en TL-Verilog y es un generador de cores con muchas opciones de configuración. Parte de este trabajo podría haber sido añadir al WARP-V la

opción de usar una NoC, al igual que existen opciones para añadir o quitar etapas del pipeline.

Rocket Chip Más que un procesador, se trata de un generador escrito en **Chisel**, lenguaje libre de definición de hardware basado en Scala que emite código sintetizable en Verilog. Ha sido diseñado por el departamento de D. Patterson, A. Waterman y K. Asanović, creadores del RISC-V. [3].

BOOM [RV64GC] *Berkeley Out-of-Order Machine*. Está creado por el mismo departamento que Rocket y con herramientas muy similares. Su característica principal es su alto grado de paralelismo a nivel de instrucción. Sin embargo, su estructura es mucho más compleja y, por lo tanto, difícil de modificar [5].

La principal ventaja de Rocket y BOOM es que, debido a su longevidad y soporte a lo largo del tiempo, cuentan con amplia documentación, literatura, herramientas, y han sido implementados tanto en FPGA como en ASICs, por lo que aunque no han sido usados directamente en este proyecto, sí que se ha consultado en ocasiones literatura relacionada con estos procesadores.

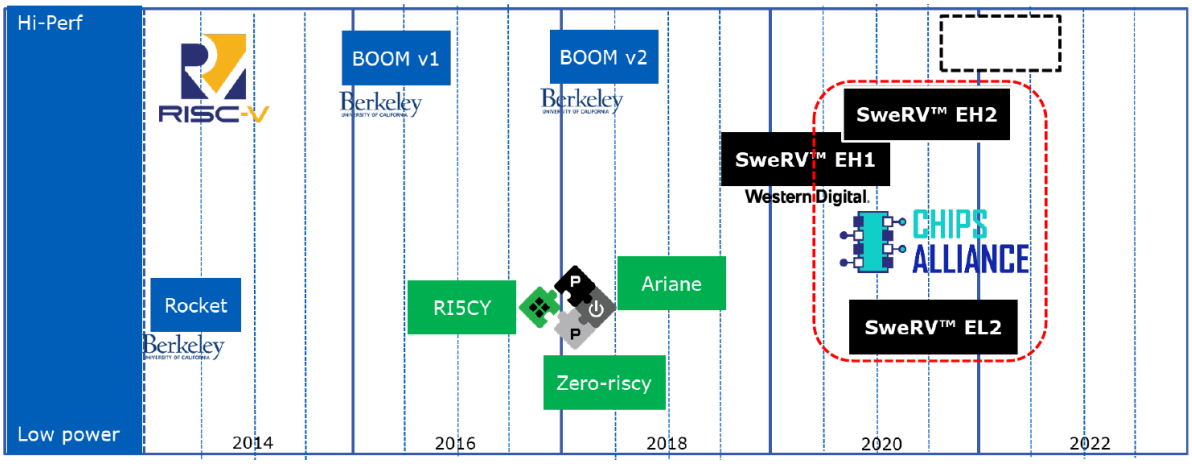


Figura 3.1.: Comparativa de la potencia de varios Cores RISC-V de código abierto. Resaltado en rojo los procesadores de la familia SweRV. Extraído de *SweRV Cores Roadmap* [4].

Finalmente, se eligió la familia SweRV debido a la familiaridad con el proyecto (se usa este procesador en la actividad formativa "Implementación ASIC de 28nm del procesador RISC V"). En la figura 3.1 se muestra una comparativa de dichos procesadores con otros cores de código abierto. Dentro de los procesadores de esta familia, se decidió modificar el EL2 por su sencillez y los pocos recursos humanos de los que se dispone en este proyecto. No obstante, tratándose de una prueba de concepto, siempre es posible en el futuro aplicar las técnicas aprendidas a otros productos más complejos, con mejoras y optimizaciones.

3.2. El procesador SweRV-EL2

El procesador SweRV-EL2 ha sido creado por Western Digital y publicado por la colaboración de código abierto CHIPS Alliance[1] en 2020. Es la segunda generación de la familia SweRV de procesadores, siendo de estos el más ligero y menos potente. Ha sido fabricado a 16 nm en TSMC con un área total de $0{,}023mm^2$, y funcionando con una frecuencia objetivo de 600MHz. Su repertorio de instrucciones es el RV32IMC+Zbb+Zbs (Zbb y Zbs son instrucciones de manipulación de bits aún no ratificadas), por lo que pueden realizarse multiplicaciones y divisiones hardware.

Además del core, el procesador cuenta con otros elementos periféricos: caché de Instrucciones, *Closely-Coupled Memory* (CCM) (tanto de datos, DCCM, como instrucciones, ICCM), interfaces de debug y AXI de 64 bits...

Su descripción de alto nivel está parametrizada con multitud de opciones de configuración que podemos fijar manualmente o ejecutando la herramienta *swerv_config_gen*, por ejemplo: si queremos optimizar para FPGA, si queremos que incluya puerto AXI, el tamaño de las cachés y sus políticas de funcionamiento, la configuración del predictor de salto... La herramienta también cuenta con cuatro perfiles objetivo predefinidos:

- **default**: Configuración por defecto con una interfaz bus AXI4.
- **default_ahb**: Configuración por defecto con una interfaz bus AHB.
- **typical_pd**: Se le quita la ICCM y tiene una interfaz bus AXI4. Es la usada para la fabricación en ASIC y la única que tiene desactivadas las optimizaciones para FPGA.
- **high_perf**: Configuración para alto rendimiento con interfaz bus AXI4. Para lograr un mejor rendimiento se aumentan algunos recursos, como el tamaño del *Branch Target Buffer* o el *Branch History Buffer*, logrando mejores predicciones de salto.

En este proyecto hemos elegido la configuración *default* con optimizaciones para FPGA, aunque no existen motivos por los que nuestras modificaciones no funcionasen con otras configuraciones.

3.2.1. Microarquitectura

El *core* del SweRV-EL2 está segmentado en 4 etapas y se emite una sola instrucción por ciclo, por lo que es *escalar*. Está diseñado para conseguir un *IPC* (Instrucciones

[1] CHIPS: *Common Hardware for Interfaces, Processors and Systems*

por Ciclo) cercano a 1,0, logrando 0,95 IPC y 0,98 IPC en las pruebas *Coremark* y *Dhrystone*.

La figura 3.2 presenta un esquema de las etapas del core. En primer lugar, se ejecuta la etapa de *Fetch*, en la que se obtiene la instrucción de la memoria y se guarda en unos registros. A continuación, se hace la decodificación (*Decode*) de la instrucción, obteniendo la información relevante: si es una instrucción de salto, una operación aritmético-lógica, los registros fuentes y destino, el inmediato codificado, etc. Después, dependiendo del tipo de instrucción puede ejecutarse uno u otro *pipe* (fase *Execute*), para continuar salvando los resultados durante la fase de *Commit*.

Las instrucciones de multiplicación pasarán por el *multiply pipe* que hace uso del multiplicador, con una latencia de 1 ciclo. Del mismo modo, las instrucciones de división pasarán por un *pipe* no segmentado que utiliza el divisor, con una latencia de 34 ciclos. El resto de instrucciones usarán el *pipe* principal —llamado *I0*—, excepto las de tipo *load* y *store*, que también cuentan con un *pipe* dedicado. En este proyecto hemos incluido una NoC que solo será usada por los *pipes* de multiplicación y división.

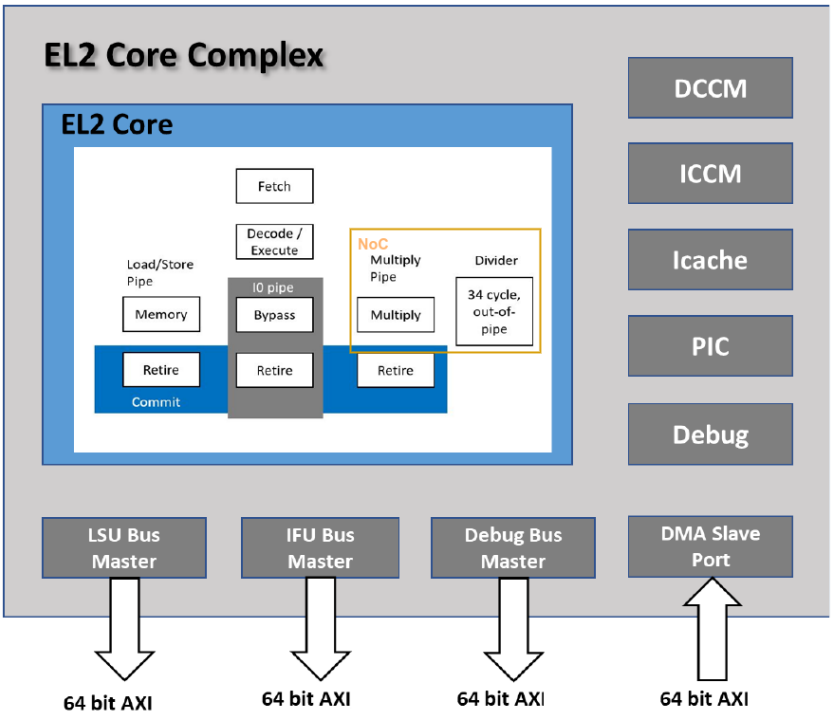


Figura 3.2.: Fases de una instrucción ejecutada en el SweRV-EL2. Señalado en naranja las fases que hacen uso de la NoC. Extraído de *SweRV Cores Roadmap* [4].

3.2.2. Diseño RTL

En cuanto al diseño RTL, programado en SystemVerilog, el *top-module* del procesador es *el2_swerv_wrapper*, que instancia y conecta la memoria (*el2_mem*) y la interfaz DMI (*dmi_wrapper*) con el *core* (*el2_swerv*).

En la figura 3.3 se muestra el diagrama de bloques del diseño simplificado y las unidades funcionales con las que cuenta el *core*, cada una con las siguientes responsabilidades:

ifu *Instruction Fetch Unit*. Unidad funcional encargada de la fase *fetch* de una instrucción. Se conecta con la memoria y obtiene la instrucción especificada por el contador del programa.

dec Se encarga de la decodificación de las instrucciones obtenidas por la *ifu*, obteniendo los operandos para la *exu* y manejando las señales de control de esta.

exu *Execution Unit*. Esta unidad funcional se encarga de ejecutar la instrucción, por lo que instancia distintos submódulos para los distintos *pipes* de ejecución. Esta es la unidad funcional a la que se añadirá la NoC.

lsu *Load-Store Unit*. Se encarga de ejecutar las instrucciones de *load* y *store*, accediendo a la memoria.

En este proyecto añadiremos una NoC para conectar los submódulos de la EXU, que se explican más en profundidad en la sección 6.2.4 Modificaciones en la unidad de ejecución.

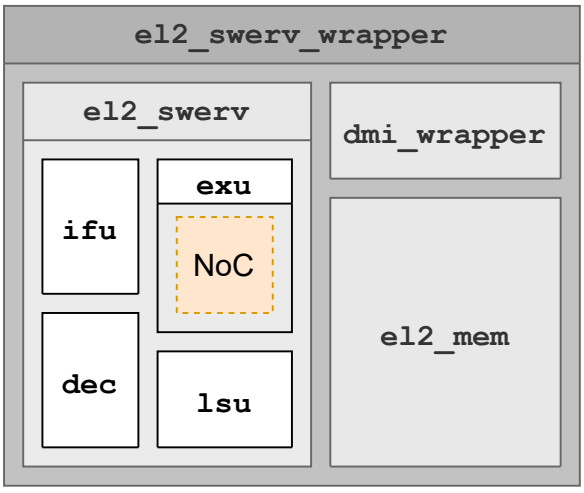


Figura 3.3.: Diagrama de bloques simplificado del diseño RTL del SweRV-EL2, mostrando las unidades funcionales del *core*. En naranja se muestra la ubicación de la NoC añadida.

Capítulo 4.

Diseño de una Red en Chip

> «A straight line may be the shortest distance between two points, but it is by no means the most interesting.»
> – Jon Pertwee como El Tercer Doctor en *Doctor Who*

En esta sección se tratan las distintas características a tener en cuenta en la especificación de una red y las decisiones de tomadas para la NoC implementada. Las definiciones usadas en esta sección están fuertemente basadas en [16, 9, 10].

Una red es un grupo de dos o más dispositivos que se enlazan entre sí para intercambiar información. Según su tamaño y el número de dispositivos interconectados, podemos clasificarlas en:

- **WANs** o redes de área amplia: funcionan alrededor del mundo y conectan miles de millones de dispositivos. Suelen interconectar distintas redes más pequeñas.
- **LANs** o redes de área local: se conectan computadores distribuidos a lo largo de una sala, edificio, o campus. Conectan miles de dispositivos.
- **SANs** o redes de área de sistema: conectan un mismo sistema, es la red usada para conectar procesadores y memorias en un supercomputador o un centro de procesamiento datos. Conectan cientos o miles de dispositivos entre sí.
- **NoCs** o redes en chip: Se conectan las unidades funcionales de una microarquitectura, dentro de un único sistema en chip. Suelen conectar pocos dispositivos, aunque puede llegar a los cientos de unidades como en el caso de las GPUs.

En nuestro caso, diseñaremos una red en chip con menos de una decena de dispositivos a interconectar, por lo que nuestro objetivo será tomar decisiones de diseño de bajo coste para reducir el impacto de la NoC en potencia y recursos consumidos.

4.1. Topología

Si queremos conectar solo dos dispositivos, podemos conectarlos directamente con un enlace punto a punto. Cuando aumentemos el número de dispositivos, podemos usar un único elemento de red (conmutador o encaminador) de N puertos para conectarlos. En el momento en el que tengamos más de N dispositivos, necesitamos usar varios elementos de red que actúen de intermediarios, conectándolos entre ellos para aumentar el número de entradas disponibles de la red en la que conectar los dispositivos o *nodos finales*.

Podemos describir la topología de una red como un grafo donde los vértices V representan los elementos de red y A los enlaces punto a punto entre ellos. Este grafo deberá ser fuertemente conexo para que todos los dispositivos tengan al menos un camino por el que llegar a cualquier otro dispositivo. El **número de saltos** es la cantidad de elementos de red por los que debe pasar un paquete para llegar a su destino, es decir, la longitud del camino que debe seguir.

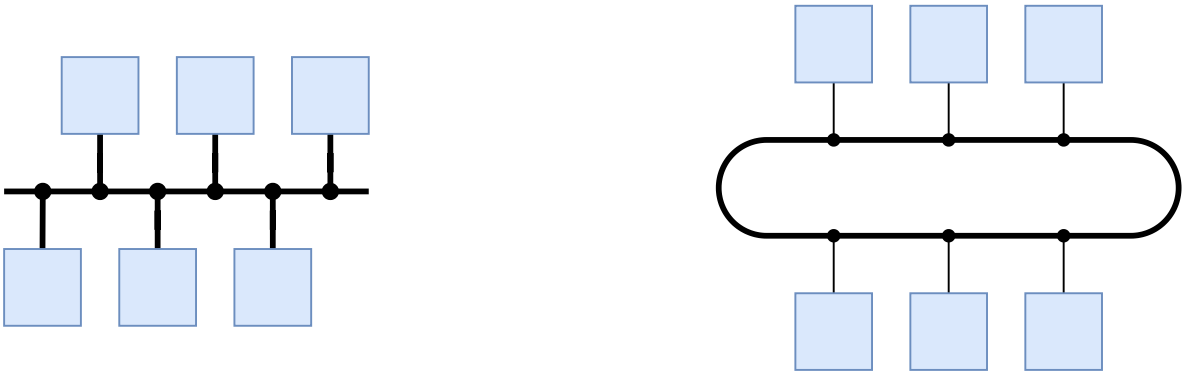
Si usamos un grafo completo, tenemos una topología en la que cualquier nodo puede acceder a distancia mínima cualquier otro nodo, pero de mucho coste (más que un crossbar), necesitando encaminadores con tantos puertos como elementos en la red. Por ello, existen alternativas que consumen menos recursos, como las topologías en **bus** y en **anillo**, en las que se conectan los nodos finales mediante un solo cable central (figura 4.1). En la topología en bus, la distancia máxima será de la longitud del bus, mientras que en la topología en anillo, esta distancia se reduce a la mitad porque el cable está cerrado, formando un ciclo.

Aumentando el número de nodos intermedios, se pueden formar topologías más complejas. Por ejemplo, con una rejilla N-dimensional de aristas conectadas cada una de ellas con su adyacente, se forma una topología en **malla** (figura 4.2a). La distancia máxima será la distancia Manhattan entre los nodos más lejanos (las esquinas opuestas), igual a la suma de la distancia de el número de vértices en los dos lados: $D = (l_1 - 1) + (l_2 - 1)$. Esta distancia máxima puede reducirse cerrando los extremos y formando un **toro**, acortándose a la suma de la división entera de los lados entre dos: $D = \lfloor l_1/2 \rfloor + \lfloor l_2/2 \rfloor$.

Según [10] y [16], dependiendo de como se conecten los nodos finales entre sí, podemos clasificar las redes de interconexión en las siguientes clases:

- **Medio compartido**: El medio es compartido entre todos los dispositivos y se necesita un método de detección o evitación de colisiones. Los buses, anillos y las redes inalámbricas serían redes de medio compartido.

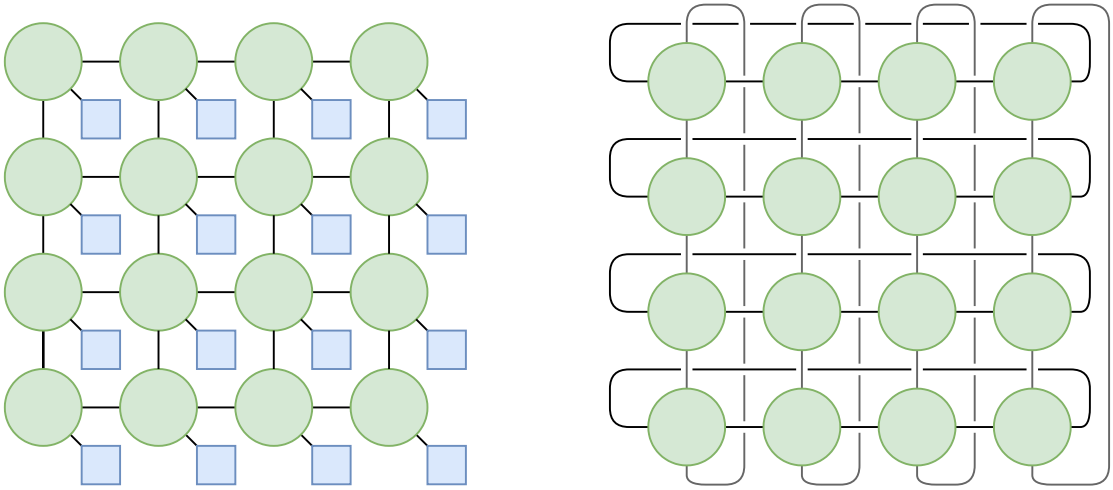
- **Medio conmutado**: Se establece dinámicamente una vía de comunicación entre la fuente y el destino.



(a) Topología en bus.

(b) Topología en anillo.

Figura 4.1.: Topologías de red en bus y en anillo. Las conexiones de los nodos finales, simbolizadas con puntos negros, pueden ser tanto conexiones a un medio compartido como nodos intermedios.



(a) Topología en malla bidimensional de 4x4. Cada nodo intermedio tiene conectado un nodo final y la distancia máxima es de 6 enlaces.

(b) Topología de toro bidimensional. Su distancia máxima es de 4 enlaces.

Figura 4.2.: Topologías en malla y toro.

- **Dinámicas/Indirectas**: Los nodos finales se conectan únicamente en los bordes de la red, lo que permite que el número de nodos sea menor al de conmutadores. Serían ejemplos de este tipo de redes los crossbars o las redes de interconexión multietapa (MINs).

- **Estáticas/Directas**: Cada encaminador está conectado directamente con un nodo emisor, no sólo en la periferia de la red, por lo que el número de encaminadores es igual al número de nodos, como en la topología en malla de la figura 4.2a

Nuestra red es una ***malla*** bidimensional de encaminadores. Aunque las mallas normalmente forman redes *directas*, nosotros hemos usado encaminadores de 4 puertos, aprovechando los extremos de la malla para conectar los nodos finales. Por lo tanto, nuestra malla sería una red *indirecta*.

4.1.1. Tipos de enlace

En las redes de medio conmutado, llamamos enlace a cada uno de los canales que conectan directamente dos dispositivos. Según el sentido de la transmisión, podemos distinguir los siguientes tipos:

- **Símplex**: Conexión unidireccional entre los dos elementos. El emisor puede enviar datos al receptor, pero no recibirlos de este.

- **Full-Dúplex**: Conexión bidireccional simultánea, en la que los dos elementos conectados pueden hacer uso de la conexión paralelamente, transmitiendo y recibiendo información al mismo tiempo.

- **Half-dúplex**: Conexión bidireccional compartida, en la que si un elemento está transmitiendo, el enlace queda ocupado. La principal ventaja es que se necesitan la mitad de recursos que full-dúplex.

En nuestra NoC nos hemos decantado por usar conexiones **full-dúplex** entre los nodos, pues las distancias entre ellos son tan cortas que el ahorro en cable sería insignificante, requiriendo implementar algún tipo de arbitraje en los enlaces para evitar colisiones cuando en un instante de tiempo los dos nodos quieran acceder al mismo.

4.2. Encaminamiento

El encaminamiento o *routing* es el proceso por el que se selecciona el camino que debe seguir un paquete por la red para llegar a su destino.

4.2.1. Problemas de encaminamiento

Durante el diseño de la red debemos ser conscientes de los siguientes **problemas de encaminamiento**:

- **Interbloqueo o *Deadlock***: Es una situación en la que un conjunto de paquetes intenta acceder a recursos que están bloqueados por otros paquetes de ese mismo conjunto, creando una dependencia cíclica imposible de resolver. Para tratar con este problema tenemos dos posibilidades:

  - **Evitación**: Podemos evitar que ocurran *interbloqueos* usando algoritmos de encaminamiento que restrinjan los caminos que toman los paquetes de tal manera que sea imposible establecer un camino que produzca un ciclo. El algoritmo más común con evitación de interbloqueos es el Encaminamiento por Orden Dimensional.

  - **Recuperación**: Si detectamos que se ha producido o puede producirse una situación de bloqueo, lo solucionamos liberando los recursos adquiridos por algunos de los paquetes que causen el problema, redirigiéndolos o cancelando su transmisión.

- **Ciclos infinitos o *Livelock***: Ocurre cuando un paquete nunca llega a su destino porque se transmite por la red indefinidamente. Una solución común para detectar *livelocks* es añadir un tiempo de vida máximo a cada paquete. También puede evitarse este problema usando algoritmos mínimos que siempre escogan el camino de menor distancia.

- **Espera indefinida, inanición o *Starvation***: Ocurre si a un paquete, a pesar de elegir un camino posible y finito, nunca consigue obtener el derecho a los recursos que solicita (p.e: si los recursos los obtienen siempre otros paquetes con mayor prioridad). Puede prevenirse implementando un arbitraje *justo*, como por ejemplo *round-robin*, pero si el arbitraje no es *fuertemente justo* siempre es posible (aunque muy poco probable) que surja este problema.

Para detectar si alguno de estos problemas aparece en nuestra NoC, se ha diseñado un modelo de simulación en SystemVerilog y se han realizado pruebas de carga.

4.2.2. Propiedades del algoritmo de encaminamiento

Podemos distinguir dos opciones según los argumentos de la función de encaminamiento que calcula la ruta a seguir:

- **Determinista**: Un paquete que entra por una entrada con un destino determinado siempre va a ser retransmitido por la misma salida, independientemente de las condiciones de la red.
- **Adaptativo**: Se tienen en cuenta las condiciones de la red para elegir el siguiente destino inmediato del paquete, pudiendo evitar congestiones, esperas indefinidas y errores en los enlaces.

Una vez elegido un tipo de algoritmo, el mecanismo de encaminamiento por el que se establece la ruta a seguir puede ser:

- **Selección de ruta en origen** (*source routing*): Se precalcula el camino a seguir en el nodo origen y se incrusta en la cabecera del paquete. Cada encaminador consumirá el primer elemento de la cabecera para saber el siguiente salto.

- **Distribuido**: Se establece en la cabecera la dirección destino, y cada encaminador se encarga de calcular el puerto de salida. Puede calcularse con una función combinacional (una *lookup table*) o con una máquina de estados.

El encaminamiento distribuido hace que cada encaminador necesite implementar la lógica de encaminamiento y, por lo tanto, ocupe más recursos. Sin embargo, la selección de ruta en origen puede aumentar considerablemente el tamaño de la cabecera y, por lo tanto, el tiempo de vuelo del paquete.

Según la distancia en número de saltos que recorrerá un paquete usando un algoritmo, este algoritmo puede ser:

- **Mínimo**: El paquete siempre recorrerá la distancia mínima necesaria para llegar a su destino. En el caso de que el algoritmo sea también adaptativo, hay que asegurarse de que elegir otro camino no creará un salto más. Los algoritmos mínimos están libres del problema de los ciclos infinitos, pero si todos los enlaces posibles están ocupados, el mensaje deberá esperar hasta que uno de estos se libere.

- **No mínimo**: El paquete no tiene por qué recorrer la distancia mínima. Puede

ser útil para evitar la congestión, pues aún recorriendo una distancia no mínima podemos tardar menos tiempo en transmitir el mensaje si evitamos esperas innecesarias usando otra ruta libre. Sin embargo, hay que intentar evitar o detectar los ciclos infinitos.

4.2.3. Direccionamiento

El direccionamiento consiste en asignar y codificar una dirección que identifique a cada uno de los elementos accesibles de la red. Si la dirección especifica la posición del receptor con respecto al emisor, decimos que el direccionamiento es *relativo*. Por otra parte, si la dirección especifica la posición de un elemento dentro de la red decimos que es *absoluto*.

En mallas N-dimensionales, el direccionamiento más sencillo es una N-tupla con las coordenadas del dispositivo (relativas o absolutas).

En nuestro caso, la dirección se especifica con una **tupla** con las dos coordenadas **absolutas** de cada elemento de la red. No se ha usado direccionamiento relativo pues sería necesario recalcular la dirección conforme el paquete va moviéndose por los encaminadores, añadiendo un sumador.

Es cierto que solo los nodos de las aristas de nuestra malla son accesibles, por lo que podría reducirse el espacio de direcciones a $2(N + M)$. Sin embargo, al codificar la dirección en bits esta mejora es menos notable: $\lceil \log_2(N + M) + 1 \rceil$ contra $\lceil \log_2(N \times M) \rceil$. Es más, en redes cuadradas el espacio de direcciones reducido empieza a ocupar menos bits a partir de 5×5.

También hemos preferido que el espacio de direcciones mantenga accesible los nodos intermedios por si es necesario cambiar la implementación para hacerlos alcanzables, sin que fuese necesario cambiar el direccionamiento ni el algoritmo de encaminamiento.

4.2.4. Encaminamiento en Orden Dimensional

El encaminamiento en orden dimensional o DOR por sus siglas en inglés (*Dimensional Order Routing*) es un algoritmo determinista, distribuido, mínimo y con evitación de interbloqueos aplicable a mallas N-dimensionales.

Para transmitir un paquete, primero se envía en una dimensión hasta llegar a la ordenada deseada, y a continuación se transmite en la siguiente dimensión. En el caso bidimensional, podemos transmitir primero el paquete horizontalmente (sólo por el eje

Y), y luego verticalmente (en el eje X) Podemos ver un ejemplo de la emisión de tres paquetes en una red bidimensional en la figura 4.3.

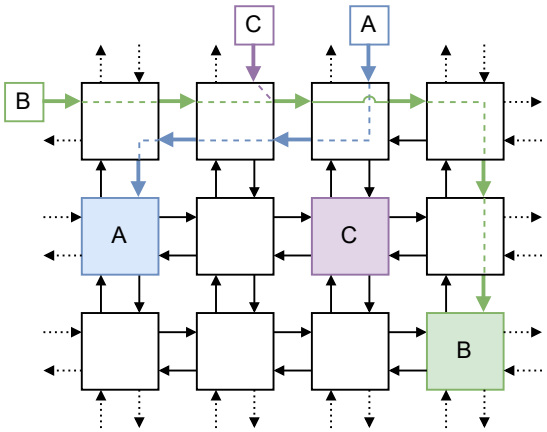


Figura 4.3.: Ejemplo de funcionamiento de DOR con prioridad vertical. Los dispositivos con fondo blanco emiten información a los nodos de fondo coloreado con la misma etiqueta. Debido a que las rutas **B** y **C** requieren de un mismo enlace, es imposible que tanto **B** como **C** lleguen a su destino. Aunque existen otros caminos mínimos disponibles, DOR no es capaz de aprovecharlos por ser un algoritmo determinista.

En un caso concreto en el que la coordenada $(0, 0)$ esté arriba a la izquierda (esquina noroeste), y con direccionamiento *absoluto*, podríamos implementarlo siguiendo el siguiente pseudocódigo:

```
def dimensional_order_routing(node_x, node_y, dst_x, dst_y) -> dir:
  if dst_y > node_y:
    return EAST
  elif dst_y < node_y:
    return WEST
  else: # dst_y == node_y
    if dst_x > node_x:
      return SOUTH
    elif dst_x < node_x:
      return NORTH
    else: # (dst_x, dst_y) == (node_x, node_y)
      return LOCAL
```

Evitación de deadlocks

Una función determinista está libre de deadlocks si y solo si no hay ciclos en su grafo de dependencias [10]. En el caso de las mallas, estos ciclos solo pueden producirse si se

realizan cuatro giros en el mismo sentido (horario o antihorario), por lo que prohibiendo uno de los cuatro giros posibles, tanto horario como antihorario, el algoritmo estaría libre de deadlocks.

En el caso de DOR con prioridad vertical, se lleva a cabo un giro únicamente cuando hemos llegado a la x deseada, por lo que solo se harán giros en los que se cambie la dirección de horizontal a vertical, eliminando la mitad de los giros potenciales y la posibilidad de ocurrir un interbloqueo.

4.3. Conmutación

La conmutación o *switching* se encarga de conectar entre sí los enlaces de la red. En el caso ideal, un enlace está solo conmutado (y ocupado) estrictamente cuando es necesario transmitir datos, maximizando el tiempo libre de cada uno. El algoritmo de arbitraje decidirá qué conmutaciones se realizan, y el elemento de conmutación de dentro del encaminador suele ser un *crossbar* o una pequeña *MIN*.

Podemos distinguir tres técnicas principales de conmutación:

- **Conmutación de circuitos**: Se reserva un camino físico antes de comenzar a enviar datos. Los enlaces usados quedarán bloqueados y no pueden ser usados por ningún otro paquete, por lo que la transmisión puede ser ininterrumpida y aprovechará al máximo el ancho de banda permitido por los enlaces. Para establecer un camino se mandan datos con la información necesaria para que el algoritmo de encaminamiento reconstruya el camino a reservar. Cuando estos datos llegan a su destino, se envía de vuelta un mensaje de confirmación, para que pueda comenzar la transmisión de los datos. Al terminar la transmisión, se pueden liberar los recursos.

- **Conmutación de paquetes**: Podemos dividir la información en paquetes de tal manera que todos ellos tengan la información del destino en una pequeña cabecera. Cada paquete es almacenado en el encaminador, que consulta la cabecera antes de reenviarlo por el puerto seleccionado. Esta técnica también es conocida como *Store-and-Forward* (SAF). Si el encaminamiento es adaptativo, los paquetes pueden llegar desordenados, por lo que será necesario introducir en la cabecera también un número de secuencia para poder recomponer el mensaje completo.

- **Virtual Cut Through** (VCT): En SAF almacenamos el paquete completo antes de realizar la decisión de enrutado. Sin embargo, una vez hemos recibido la información del destino ya podríamos tomar la decisión de encaminamiento y comenzar a reenviar la información, disminuyendo considerablemente la latencia cuando no

se producen bloqueos. Sin embargo, sigue siendo necesario tener búfers que almacenen el paquete si se produce un bloqueo.

- **Wormhole**: Se hace algo parecido a VCT, sin embargo, para evitar almacenar todo el paquete cuando ocurran problemas, se dividen los paquetes en *flits*[1]: unidades indivisibles que son almacenadas. De este modo, no es necesario almacenar todo el paquete, por lo que se reduce el tamaño de los búfers de cada encaminador. Al estar cada paquete "segmentado", puede ocurrir que en un instante el paquete esté almacenado en varios switches distintos.

La NoC implementada hace uso de una técnica basada en las ideas de conmutación de circuitos y wormhole, en la que se dividen los paquetes en *flits* y el primero de ellos (llamado cabecera) se encarga de enviar los datos para realizar la conmutación de circuitos [20, 12].

4.3.1. Arbitraje

El arbitraje es la política por el que se decide qué paquete tendrá acceso a qué recursos (los enlaces disponibles) en caso de conflictos. Para evitar la inanición cuando dos o más paquetes pidan un mismo recurso, el algoritmo de arbitraje deberá ser *justo*, por ejemplo arbitrando por orden de llegada, o con *round-robin*.

Puede implementarse algún sistema de prioridad si en la cabecera se incluye información adicional, haciendo que los mensajes más importantes pasen primero.

En nuestra NoC, el arbitraje usa *round-robin*.

4.4. Diseño de los flits y paquetes

Llamamos *flit* a cada una de las unidades lógicas mínimas que podemos enviar por la red. Un paquete está formado por varios *flits*, donde el primero de ellos solo tiene información relevante para los conmutadores y llamamos *cabecera*, y el último se encarga de cerrar la conexión y se denomina *cola* (figura 4.4). Mandamos los flits de manera secuencial para formar un paquete, sin embargo, el envío de cada uno de los flits individuales puede realizarse tanto de manera secuencial (segmentada) como en paralelo con un bus del tamaño del flit.

[1]La palabra *flit* es un acrónimo de *flow control unit* o unidad de control de flujo.

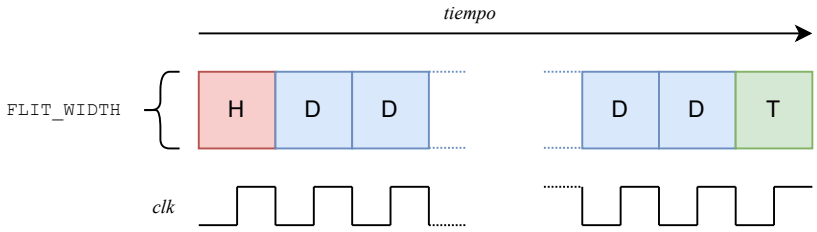


Figura 4.4.: Diagrama temporal del envío de un paquete completo a lo largo de la red.

Una alternativa al sistema de *flit de cola* es especificar el tamaño del paquete en la cabecera. Sin embargo, esta solución requiere de aumentar el tamaño de la cabecera y restringe el tamaño máximo de un paquete.

En caso de querer enviar un paquete de un tamaño en bits que no sea múltiplo del ancho del flit, es necesario comunicar de alguna manera el tamaño en bits del paquete para poder ignorar los sobrantes. Por ejemplo, especificándolo en los bits libres de la cabecera, o fijándolo en el protocolo si el tamaño es invariable y conocido de antemano.

En nuestra NoC, cada *flit* se manda en paralelo por un bus de tamaño *FLIT_WIDTH* para evitar crear más dominios de reloj y simplificar el diseño. En caso de enviar los bits en serie se necesitaría una frecuencia de reloj demasiado alta (aumentando el consumo dinámico), mientras que añadir pistas al diseño no supone un problema.

El flit cabecera contiene simplemente la información de destino del paquete, y los flits de datos y cola tienen casi todo el espacio libre. Puede verse de manera detallada la codificación de los flits en la subsección 4.5.1

4.5. Resumen de la especificación de la NoC

La NoC de este proyecto cuenta con las siguientes características:

- **Topología** en *malla* de altura y ancho parametrizables.
- **Enlaces** *full-dúplex*.
- Algoritmo de **encaminamiento** en *Orden Dimensional* (DOR), siendo este determinista y mínimo.
- **Direccionamiento** mediante una tupla con las *coordenadas absolutas* del dispositivo.
- **Conmutación** de circuitos segmentada usando *flits* enviados en serie.
- **Arbitraje** *round-robin*.
- Cada *flit* se manda en paralelo usando un *bus*.

- Sistema de **paquetes** con tres tipos de *flits*: cabecera, datos y cola.

A nivel usuario, para poder crear dispositivos que hagan uso de la red, también deben tenerse en cuenta la codificación de los flits (4.5.1) y el protocolo de comunicación (4.5.2).

4.5.1. Codificación de los flits

De un flit de tamaño *FLIT_WIDTH*, los dos primeros bits han sido reservados para un enumerado *e_flit* que representa el tipo de flit, pudiendo ser cabecera (*HEADER*), datos (*DATA*) o cola (*TAIL*). Dependiendo del tipo de flit, podemos distinguir las siguientes responsabilidades y formatos:

Cabecera Se encarga de establecer el camino virtual, por lo que contiene la dirección destino del paquete. Los bits que no tienen ningún uso especificado pueden usarse para transmitir información si es necesario.

Datos Podemos enviar tantos flits de datos como sea necesario, pero se recomienda minimizar el tamaño del paquete para evitar la *inanición* de otros paquetes de la red, pues al usar conmutación de circuitos los canales usados quedan reservados mientras se envia el paquete. Si descontamos los dos bits del enumerado, tenemos *FLIT_DATA_WIDTH* bits para enviar información.

Cola La única diferencia con los flits de tipo *DATA* es que, cuando un nodo recibe un flit de cola, este debe liberar los recursos reservados para la conexión. Por lo tanto, solo se enviará un flit de cola al final del paquete.

Podemos ver una representación visual del formato de los flits en la figura 4.5.

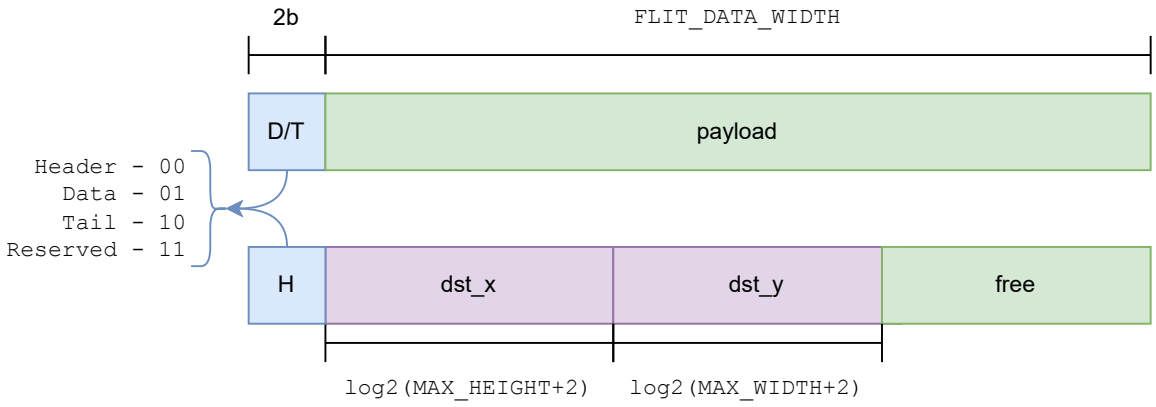


Figura 4.5.: Formato de los flits de la red. Arriba el formato de los flits de tipo datos/cola y abajo el formato de los flits cabecera.

4.5.2. Protocolo de comunicación

En esta sección se especifica el sistema de reglas que deben seguir los usuarios de la red para transmitir y recibir información satisfactoriamente. Pueden verse ejemplos de implementación y máquinas de estado en las subsecciones 6.2.1 Emisor y 6.2.2 Receptor.

En la comunicación entre nodos, junto al flit se envía una señal de control *enable* y se recibe una respuesta *ack*:

- ***enable***: indica al siguiente nodo que el bus del flit contiene información y es válida.
- ***ack***: indica al nodo anterior que se ha establecido un camino virtual y puede comenzar el envío de datos.

Usando estas dos señales y el bus de flit, la comunicación según si queremos emitir o recibir datos sería la siguiente:

Emisión

Para establecer una conexión es necesario crear una cabecera indicando la dirección de destino con el formato estipulado en el apartado anterior. A continuación, habilitamos el enlace poniendo la señal *enable* a 1, y continuamos repitiendo el flit cabecera mientras se reserva el camino completo hasta recibir la señal *ack*; que indica que se ha establecido el camino virtual.

Desde este momento, la señal *ack* se mantendrá a 1 y podemos enviar tantos flits de tipo *DATA* como sea necesario. Cuando deseemos mandar los últimos datos y liberar los recursos, mandaremos un flit de tipo *TAIL*, finalizando la conexión.

Recepción

Si podemos recibir paquetes, es necesario establecer la señal *ack* a 1. Cuando vayamos a recibir un paquete se habilitará la señal de entrada *enable* a la vez que recibimos una cabecera. Esta cabecera podemos descartarla o almacenarla si los bits libres tienen datos de nuestro interés. Durante varios ciclos recibiremos de nuevo esta cabecera con exactamente los mismos datos, por lo que debemos descartarla hasta recibir un flit de tipo *DATA*.

Los siguientes bits serán de tipo *DATA* o *TAIL* y contendrán la información del paquete, por lo que los procesaremos como sea necesario. Finalmente, recibiremos un flit de cola que indicará el fin de la conexión.

Parte III.

Diseño e implementación

Capítulo 5.

Tecnologías y metodologías usadas

«Always pass on what you have learned.»
– Master Yoda en *Return of the Jedi*

En las herramientas, se han usado principalmente herramientas compatibles con las FPGAs que posee el grupo de investigación, fabricadas por Xilinx, el mayor fabricante de FPGAs.

Por otro lado, se han utilizado lenguajes de scripting como *Python*, *Bash*, *Perl* o *TCL* para la realización de distintas tareas. El uso de algunas de las herramientas se ha realizado mediante una conexión por escritorio remoto a una máquina Windows con las herramientas preinstaladas, debido a los altos requisitos del sistema para la ejecución de este tipo de aplicaciones.

En las siguientes secciones se detallan las herramientas más usadas en el proyecto.

5.1. Redacción y documentación

5.1.1. Overleaf y LaTeX

Para la redacción de la memoria de este proyecto se ha usado el sistema de preparación de documentos LaTeX. Un lenguaje de marcado en el que se intercala redacción en texto plano con macros que especifican su formato.

Para hacer posible la colaboración entre los integrantes del proyecto (estudiante y directores) se ha usado Overleaf, un editor colaborativo alojado en la nube que usa el conjunto de paquetes de LaTeX “TeXlive”para proporcionar un entorno completo de elaboración de documentos sin necesidad de instalar nada.

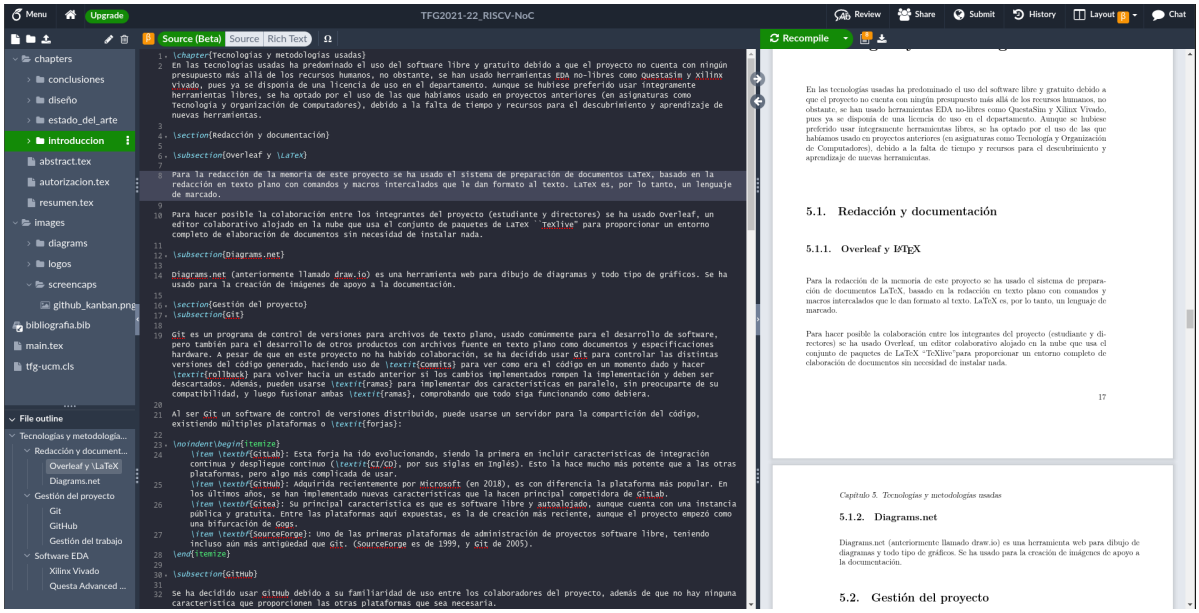


Figura 5.1.: Captura de pantalla del proyecto de la memoria en Overleaf

Para la creación de diagramas e imágenes de apoyo a la documentación se ha usado diagrams.net (anteriormente llamado draw.io), una herramienta web para el dibujo de todo tipo de gráficos.

5.2. Gestión del proyecto

5.2.1. Git

Git es un programa de control de versiones para archivos de texto plano, usado comúnmente para el desarrollo de software, pero también para el desarrollo de otros productos con archivos fuente en texto plano como documentos y especificaciones hardware. A pesar de que en este proyecto no ha habido colaboración, se ha decidido usar Git para controlar las distintas versiones del código generado, haciendo uso de *Commits* para poder visualizar las diferencias en el código a lo largo del tiempo y hacer *rollback* para volver hacia un estado anterior si los cambios implementados rompen el correcto funcionamiento del proyecto y deben ser descartados. Además, pueden usarse *ramas* para implementar dos características en paralelo, sin preocuparte de su compatibilidad, y luego fusionar ambas *ramas*, comprobando que todo siga funcionando como debiera.

GitHub

Al ser Git un software de control de versiones distribuido, puede usarse un servidor para la compartición del código, existiendo múltiples plataformas o *forjas*. Además de compartir código, la mayoría de forjas tienen una interfaz web en la que se dan herramientas para facilitar la colaboración. De entre ellas, se ha elegido **GitHub** debido a la familiaridad de uso entre los colaboradores del proyecto. En los últimos años han implementado diversas características y mejoras a la gestión de proyectos, haciendo que, además de alojar el código, pueda llevarse a cabo la dirección del proyecto y asignación de tareas desde una única plataforma.

Se han creado múltiples repositorios para manejar mejor las distintas partes diferenciadas del proyecto.

- **daviddavo/Cores-SweRV-EL2-NoC**: Un *fork* o ramificación del repositorio del SweRV-EL2 de chipsalliance. En este repositorio se han realizado los cambios tanto en el diseño como en el sistema de compilación para incluir la NoC en la unidad de ejecución.

- **daviddavo/tfg_poc_noc**: Se incluye el código del diseño y herramientas para la compilación de la NoC de manera aislada.

- **ogarnica/TFG2021-22_RISC-V_NoC**: Proyecto principal. Cuenta con los otros dos proyectos instalados como *submódulos*, de tal manera que al descargar este proyecto se descargan también los otros dos. Entre los archivos del proyecto también se encuentra documentación técnica, el código LaTeXde esta memoria, los datos usados en el procesamiento de datos para los gráficos de la memoria, y el sistema de issues y kanban usados para la gestión del proyecto.

5.2.2. Gestión del trabajo

Para la gestión de tareas se ha usado un tablero *Kanban*, en el que cada tarea está representada por una tarjeta que se va moviendo entre las diversas columnas dependiendo de su grado de completitud. Cada una de las tarjetas permite hacer un seguimiento exhaustivo de la tarea que representa, pudiendo ser etiquetada, asignada a una persona, y comentada. En nuestro caso, el tablero cuenta con las siguientes columnas:

- ***To do***: Se incluyen todas las tareas por hacer o a considerar, pueden no estar asignadas a nadie aún.

- ***Sprint***: Tareas por hacer, pero con un plazo de entrega próximo. Son tareas que reciben una especial prioridad y deben completarse lo antes posible.

- ***Blocked***: Tareas bloqueadas a la espera de que acabe algún otro proceso. Pueden estar bloqueadas por otra tarea del proyecto, o por una causa externa (por ejemplo, esperar a que la licencia de un software se renueve).

- ***In progress***: Tareas que se están realizando ahora mismo. En la filosofía Kanban es muy importante poner un límite al número de tareas que puede manejar un solo usuario, y poder visualizar en todo momento las tareas que se estan realizando actualmente.

- ***Review***: Tareas terminadas que deben ser revisadas. En este caso, por los directores del proyecto.

- ***Done***: Esta columna sirve de registro de todas las tareas que han sido terminadas.

Este tablero ha sido alojado también en GitHub, permitiéndonos hacer referencias al código fácilmente y trabajar con las ramas de git.

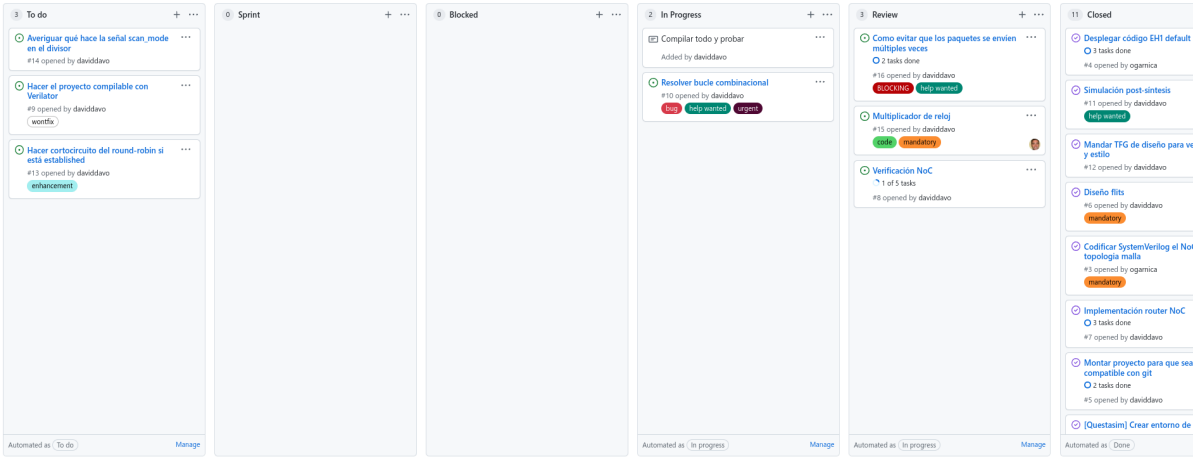


Figura 5.2.: Captura de pantalla del tablero Kanban de GitHub en la fase tardía del desarrollo del proyecto

5.3. Diseño hardware

El software *Electronic Design Automation* (EDA) o de Diseño Electrónico Automático, también llamado *Electronic Computer-Aided Design* (ECAD) o de Diseño Electrónico Asistido por Computadora, sigue un flujo de diseño para la descripción, elaboración y análisis de circuitos integrados.

Por lo general, el flujo de trabajo es el siguiente (resumido en la figura 5.3):

- **Diseño**: Se escribe en algún lenguaje de descripción de hardware (como SystemVerilog o VHDL) el diseño, creando una representación del circuito en alto nivel conocida como RTL (*Register-Transfer Level*). En el RTL se modelan los registros y las operaciones lógicas de las señales que circulan entre ellos. Antes de continuar, debemos ejecutar una simulación de este modelo, creando un *testbench* para verificar si el diseño cumple con las funcionalidades requeridas (también llamada simulación *conductual* o de *comportamiento*).

- **Síntesis**: Es un proceso en el que se traduce el RTL a una *netlist*: una descripción que representa las puertas lógicas o registros utilizados y las conexiones entre ellos. También se generan reportes con estimaciones temporales y de consumo, y de problemas combinacionales que puedan surgir (por ejemplo, si el circuito tiene bucles combinacionales, o se puede calcular el camino más largo posible que restringirá nuestro ciclo de reloj). Finalmente, podemos realizar una simulación post-síntesis en la que se añade cierto retardo aproximado a cada una de las puertas y conexiones, comprobando que el funcionamiento sigue siendo el esperado.

- **Implementación**: Se optimiza el *netlist* generado en el punto anterior y se asigna a cada elemento de la *netlist* una de las primitivas de la FPGA (*map*), por ejemplo, para implementar una función lógica se usarán varios LUTs distintos, pero sin tener en cuenta su ubicación en la FPGA. En el *place & route* se coloca cada uno de los elementos de la netlist en un elemento físico de la FPGA, intentando cumplir las restricciones temporales establecidas. Finalmente, se traduce esta información a un binario que configurará la FPGA (*bitstream*). En este moneto ya podemos estimar con precisión el tiempo que tardarán las operaciones combinacionales, su consumo, o incluso la temperatura alcanzable por las áreas de la FPGA. Con herramientas más avanzadas puede comprobarse que los campos electromagnéticos generados por las señales no generarán problemas como *crosstalk*, o simularse la lógica a nivel de transistor en caso de querer fabricar el circuito.

En este proyecto nos hemos enfocado en la fase de diseño RTL, pero creando código que sea *sintetizable* para que pueda ser usado en una FPGA si es preciso.

5.3.1. FPGA

Una Matriz Programable de Puertas Lógicas o FPGA (*Field-Programmable Gate Array*) es un dispositivo diseñado para permitir que el usuario configure la matriz de puertas lógicas de tal manera que pueda formarse un circuito.

La FPGA contiene bloques lógicos configurables (CLBs) que, conectados y configurados

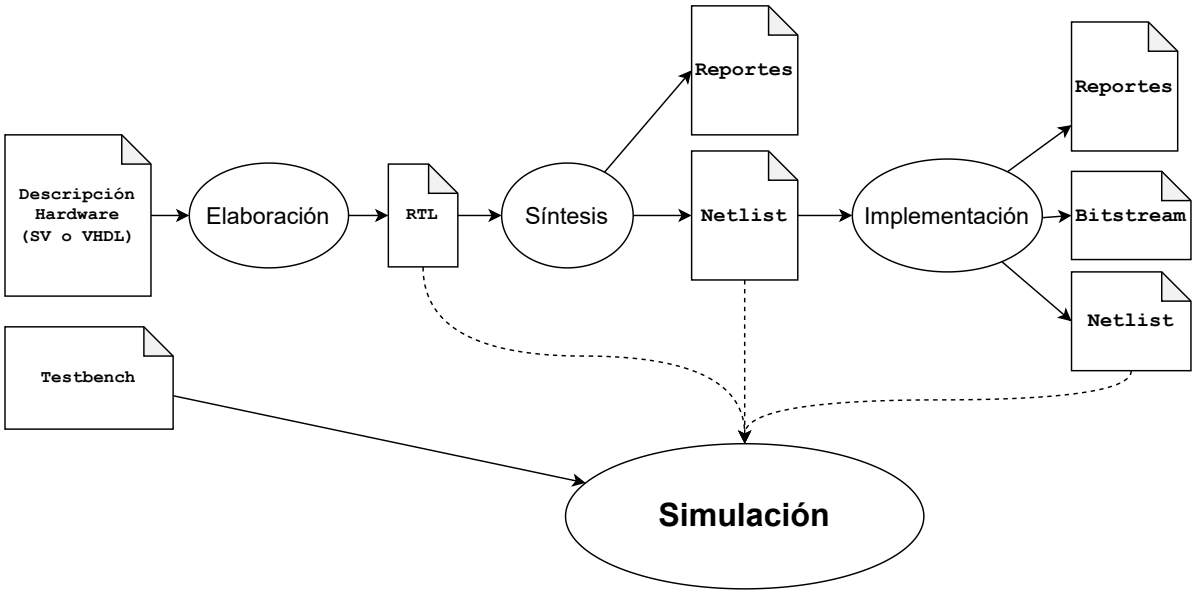


Figura 5.3.: Representación del flujo de diseño hardware.

correctamente, nos permiten crear cualquier circuito lógico. Cada CLB cuenta con varios elementos, como registros o *flip-flops* y tablas de verdad o LUTs (*LookUp-Tables*), que permiten especificar cualquier función lógica (normalmente, de un número reducido de entradas, 3 o 4 bits). Algunos CLBs pueden incluir funciones lógicas más complicadas pero muy usadas, como sumadores o multiplexores. Aun así, algunas FPGA incluyen elementos fuera de los CLBs accesibles por nuestro circuito, como son memorias de acceso aleatorio en bloque (BRAM), procesadores de señales digitales (DSP) o interfaces de comunicación avanzadas (Ethernet, PCIe).

También es posible reconfigurar tan solo partes de la FPGA, permitiéndonos modificar parte del diseño en tiempo de ejecución, añadiendo, modificando, quitando, intercambiando o reubicando módulos cuando sea necesario; por ejemplo, si un diseño es demasiado grande puede dividirse en submódulos que se van llevando a la FPGA según se van necesitando. Además, esta característica puede aprovecharse para crear diseños tolerantes a fallos: en caso de que la radiación electromagnética degrade el sustrato en el que se sitúa un submódulo, este puede reimplementarse en un área libre de la FPGA para que el dispositivo siga funcionando correctamente.

5.3.2. Xilinx Vivado

Vivado es el software EDA elegido para el diseño y la síntesis de la NoC. La versión utilizada es la 2021.2, con la licencia WebPACK Edition, sin coste pero con una cantidad disponible de dispositivos objetivo reducida. Se ha elegido Vivado pues al estar creado por

Xilinx, la misma compañía que produce la FPGA elegida, tiene una gran compatibilidad con esta. Además, no ha sido necesario aprender su uso al haberse usado anteriormente en otras asignaturas del grado.

Es capaz de mostrar los diagramas de bloques del RTL generados, muy útil para la depuración y solución de los errores detectados durante la síntesis y simulación. También pueden generarse multitud de reportes, presentando los recursos consumidos por nuestro diseño en potencia y área de la FPGA, o señalando violaciones de las restricciones temporales especificadas para las señales de nuestro diseño (por ejemplo, si un reloj tiene demasiada latencia puede hacer que se desincronice la lógica secuencial).

Sin embargo, el simulador integrado en Vivado no cuenta con demasiadas características y no es muy potente, por lo que se ha optado por usar un simulador externo.

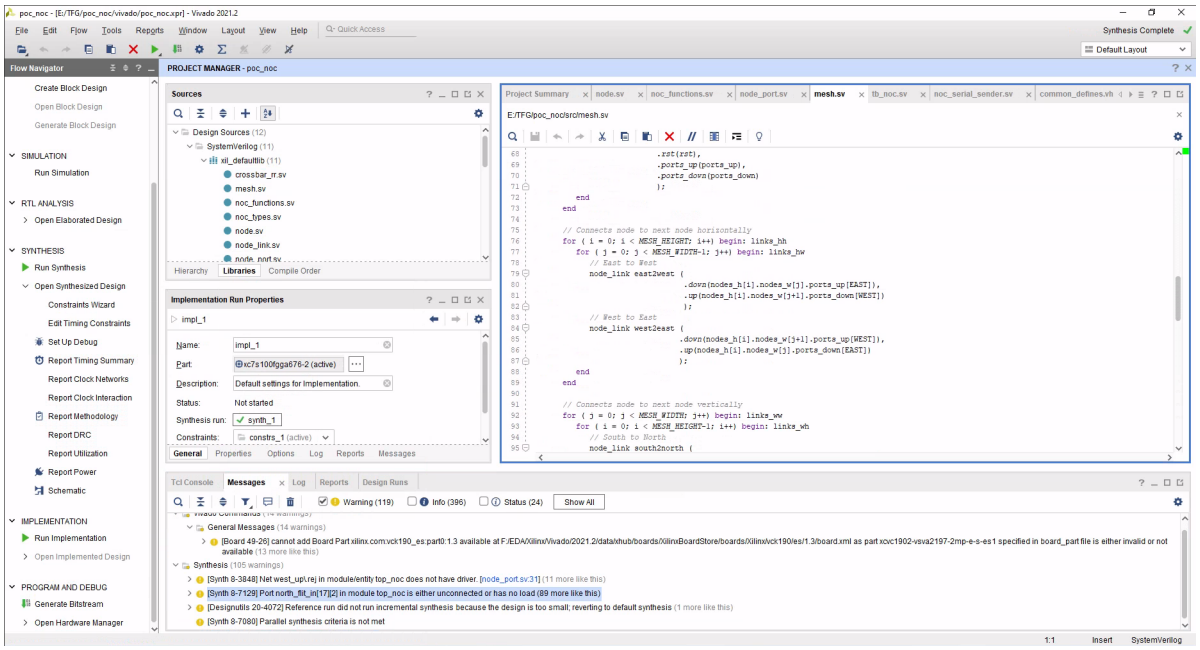


Figura 5.4.: Captura de pantalla de Xilinx Vivado 2021.2 en su vista de edición de código

5.3.3. Questa Advanced Simulator

QuestaSim es un entorno gráfico de verificación, simulación y depuración de lenguajes de descripción de hardware de Siemens EDA[1]. Actualmente en el proyecto se utiliza la versión 2022.1. Se ha usado para simular y depurar los diseños RTL y los sintetizados por Vivado.

[1]La productora original del simulador es Mentor Graphics, adquirida en 2017 por Siemens

Cuenta tanto con un entorno por línea de comandos para realizar pruebas *por lotes* no supervisadas (pudiendo exportar los resultados de las formas de onda), como con una interfaz gráfica en la que es posible mostrar un diagrama de tiempos con las formas de onda de las señales de nuestro diseño, tanto de entrada/salida como internas.

Puede integrarse con Vivado para ejecutar fácilmente simulaciones post-síntesis utilizando las librerías de simulación de los elementos básicos de la FPGA, incluyendo los retardos de cada una de las señales y puertas lógicas.

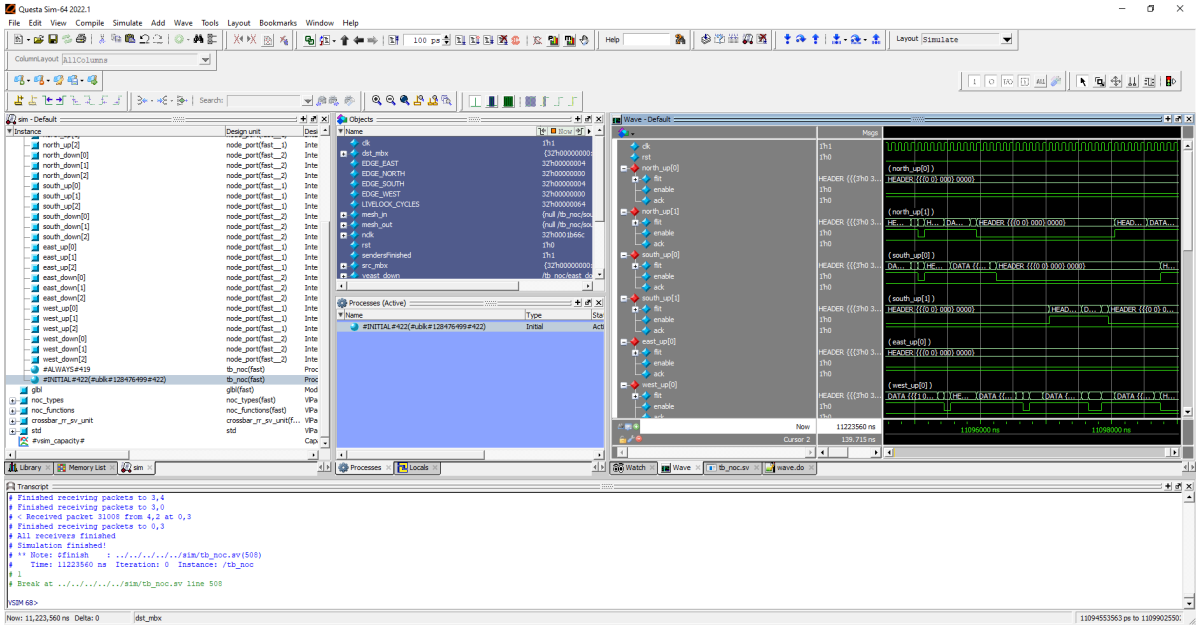


Figura 5.5.: Captura de pantalla de QuestaSim 2022.1 tras ejecutar una simulación de la NoC.

Capítulo 6.

Diseño RTL

«Don't panic»
– La Guía del Autoestopista Intergaláctico

Se ha generado un diseño RTL escrito en SystemVerilog con dos partes diferenciadas: una NoC autónoma parametrizable que puede ser conectada en cualquier sistema, y un conjunto de módulos que adaptan la NoC a la unidad de ejecución (EXU) del procesador SweRV-EL2.

Los submódulos de la EXU utilizan la NoC como intermediaria, para enviar las señales de entrada de la EXU al submódulo correspondiente, y las de salida del submódulo a la salida de la EXU.

Para el diseño e implementación del circuito se ha seguido una estrategia de abajo a arriba o *bottom-up*, comenzando con el diseño e implementación de las partes individuales y, una vez simuladas y verificadas, continuando con las partes más grandes que instancian dichos módulos.

6.1. Diseño de la Red en Chip

Se ha especificado una NoC con las características de la sección 4.5. La figura 6.1 ilustra un diagrama con los distintos módulos y su jerarquía, siendo los siguientes:

- **`mesh`**: Es la malla encargada de conectar los distintos encaminadores (`router`) entre sí, y de proveer un array de puertos de entrada y salida. Es el módulo *top* de la NoC.
- **`router`**: Implementa el encaminador, recibe los paquetes de los `router`s o dispositivos adyacentes, los procesa, y los reenvía por el puerto correspondiente, haciendo uso de un `crossbar`.

- `crossbar`: Se encarga de la conmutación, permitiendo que el `router` realice el encaminamiento.

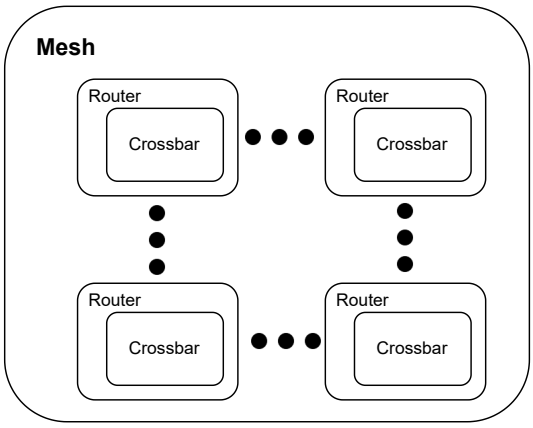


Figura 6.1.: Bosquejo de la jerarquía de módulos de la NoC, siendo `mesh` el *top-module*, y pudiendo instanciarse los `router` todas las veces que se especifique en los parámetros de la `mesh`.

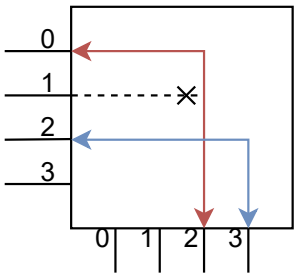
En nuestro caso, siguiendo la metodología *bottom-up* hemos diseñado en primer lugar el `crossbar`, luego el `router`, y después el `mesh`. Aun así, durante el diseño de los módulos superiores han surgido nuevos requisitos de los módulos inferiores, por lo que su especificación no ha sido completada hasta finalizar el proyecto.

Durante la implementación se ha intentado parametrizar los diseños para facilitar la reusabilidad de los módulos y reducir la cantidad de código duplicado.

6.1.1. Crossbar

El objetivo de un `crossbar` es conectar n entradas con n salidas sin contención. El `crossbar`, por lo tanto, recibirá peticiones para conectar un puerto *fuente* con uno *destino* y deberá decidir si cumplirlas o denegarlas. Además, el `crossbar` implementado conecta también un camino de *back-propagation* que permite enviar datos desde el puerto *destino* al *fuente*. Si dos puertos piden el mismo destino, se le concederá la conexión únicamente a aquel que tenga mayor prioridad.

Por ejemplo, en un `crossbar` de 4 puertos, si las fuentes *0* y *2* intentan establecer una conexión con los destinos *2* y *3*, no se producirá ningún problema y ambas peticiones podrán ser satisfechas. Sin embargo, si a esta situación le añadimos que el puerto *1* pide conectarse a la salida *2*, habrá un conflicto con la petición de *0*, con lo que el `crossbar` deberá decidir cual de las dos peticiones cumplir. La figura 6.2 ilustra este ejemplo en el que se ha decidido llevar a cabo la petición de 0 y denegar la de 1.



p	dest[p]	ack[p]
0	2	1
1	2	0
2	3	1
3	-	0

Figura 6.2.: Ejemplo de establecimiento de conexiones de un `crossbar` de cuatro puertos. En la tabla de la derecha, `p` es el número de puerto, `dest[p]` el puerto destino pedido por `p`, y `ack[p]` la respuesta del `crossbar` a esa petición. Como el puerto `0` y el puerto `1` piden la misma salida, solo se establece la conexión con el de mayor prioridad (el `0`).

La prioridad, en lugar de estar fija, se establece mediante *round-robin* para evitar el problema de la *espera indefinida* (Subsección 4.2.1). El `crossbar` tiene un contador módulo n –el número de puertos– que se aumenta cada ciclo de reloj e indica cual es el puerto de mayor prioridad.

El `crossbar` está parametrizado por los siguientes parámetros:

- **PORTS**: Número de puertos de entrada y salida del `crossbar`.
- **WIDTH**: Anchura en bits del canal de conexión de la fuente al destino.
- **BP_WIDTH**: Anchura en bits del canal de conexión inverso (*back-propagation*).

Teniendo en cuenta todas las restricciones, y los parámetros, la tabla 6.1 presenta los puertos del `crossbar`.

Dir.	**Tamaño**	**Puerto**	**Cantidad**	**Comentario**
I	1	clk	1	Reloj de la NoC para el round-robin
I	1	rst	1	Reset para el registro del round-robin
I	WIDTH	data_i	PORTS	Los distintos canales de entrada
I	BP_WIDTH	bp_data_i	PORTS	Entrada del back-propagation
I	$\log_2 PORTS$	dest	PORTS	Petición del puerto de salida
I	1	dest_en	PORTS	Si la petición es válida o no
O	WIDTH	data_o	PORTS	Datos de salida por cada puerto
O	1	data_o_en	PORTS	Si los datos de salida son válidos
O	1	bp_o	PORTS	Salida del camino de back-propagation
O	1	ack	PORTS	Si se ha asignado la petición de destino

Tabla 6.1.: Puertos del *Crossbar*

6.1.2. Encaminador

El módulo `router` es el elemento de la red encargado de realizar el encaminamiento de la información. El `router` no es más que una pequeña unidad de control que maneja un `crossbar` de 4 puertos del ancho del *flit*. Cada `router` tiene 4 puertos (Norte, Sur, Este y Oeste) y, por cada puerto, una máquina finita de estados o FSM que representa el

estado de la conexión por ese puerto. Como las decisiones de encaminamiento se toman por cada puerto, decimos que el encaminamiento es *distribuido* en lugar de centralizado. La figura 6.3 muestra el diagrama de transición de estados de la FSM usada, con los siguientes estados:

- ***IDLE***: Aún no se está enviando información por este puerto. En el caso en el que nos llegue una cabecera, de manera combinacional aplica *Dimensional Order Routing* para calcular el puerto de destino y comprueba en los registros de destino de los otros puertos si ningún otro `router` lo está usando. En caso de que entren dos cabeceras a la vez con el mismo destino, el `crossbar` deberá encargarse de realizar el arbitraje. El puerto para el que el `crossbar` haya cumplido la petición pasará al estado *ESTABLISHING*, guardando la información de destino en un registro y bloqueando la salida por ese puerto.

- ***ESTABLISHING***: Como el establecimiento del circuito virtual está segmentado, la cabecera debe seguir propagándose por los nodos intermedios hasta el ciclo en el que se reciba un *ack* por su puerto de *back-propagation*, momento en el cual podemos pasar a enviar la información.

- ***ESTABLISHED***: La conexión ha sido establecida y solo es necesario mantener la conmutación, hasta que se reciba un flit de tipo *TAIL*, que liberará los recursos haciendo que el puerto pase al estado *IDLE* de nuevo.

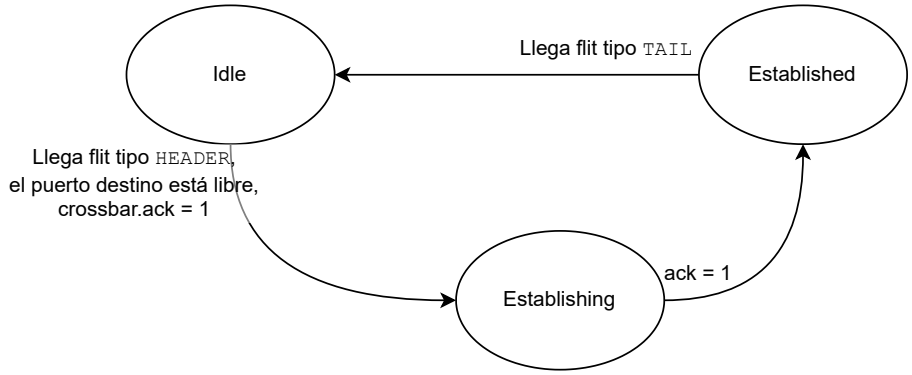


Figura 6.3.: FSM del puerto de un `router`.

Para que sea posible realizar el encaminamiento, debido a que el direccionamiento es *absoluto*, el `router` debe recibir por parámetro su posición (x, y) dentro de la red. Además, para evitar mandar paquetes por el *borde* (donde no hay `routers`), recibe la posición máxima hasta la que puede encaminar.

Las entradas y salidas del encaminador son simplemente el reloj de la NoC, una señal de reset global asíncrona para los registros de la FSM, y 4 puertos de entrada y 4 de salida del tipo interfaz *node_port*, explicado en la siguiente subsección.

6.1.3. Interfaz puerto

En SystemVerilog, una interfaz es una manera de encapsular un conjunto de señales relacionadas en un bloque, para que pueda ser instanciada y reusada a lo largo del proyecto, facilitando el desarrollo y minimizando los problemas al realizar cambios [7].

Para codificar el tipo de las entradas/salidas de los puertos de los elementos de la red se ha decidido usar una *interfaz* de SystemVerilog. De esta manera, al crear un módulo que use dichas señales, tan solo es necesario poner como puerto esta interfaz. Por ejemplo, si se añadiese una nueva señal de control a la comunicación, no sería necesario cambiar la descripción del módulo `mesh`.

Las interfaces de SystemVerilog permiten definir *modports*, en los que se especifica la dirección de las señales. Una misma interfaz puede tener varios *modports* para especificar las distintas "perspectivas" que pueden tener distintos componentes al usar esa interfaz.

En nuestro caso, se define un *modport down* (de *downlink* o *download*) que usarán los dispositivos que puedan recibir información de la red. Los dispositivos que deseen enviar información deben declarar el puerto usando el *modport up* (de *uplink* o *upload*), en el que se cambia la dirección de todas las señales.

Finalmente, en la tabla 6.2 se muestran las señales que definen la interfaz `node_port`, desde el *punto de vista* de un dispositivo que recibe un flit (modport *down*).

Dir.	Tipo	Nombre	Descripción
I	flit_t	flit	El flit a enviar
I	logic	enable	Si queremos habilitar la conexión (la información en *flit* es válida)
O	logic	ack	Si se ha conseguido establecer una conexión

Tabla 6.2.: Puertos de la interfaz `node_port` con modport *down* (entrada de datos).

6.1.4. Malla

El módulo `mesh` instancia los distintos `router` y los conecta entre ellos formando una malla bidimensional. Sus entradas son un reloj y un reset global asíncrono que será pasado a cada uno de los nodos. Además, expone arrays de interfaces de tipo `node_port` a lo largo de su perímetro, por lo que tiene un total de $2 \cdot (WIDTH + HEIGHT)$ `node_port`s de entrada (*down*), y otros tantos de salida (*up*).

Para ello, recibe por parámetro el ancho y alto deseado y, usando bucles *generate* de SystemVerilog, instancia los nodos y sus conexiones. Al igual que se creó la interfaz `node_port` para facilitar la implementación de la malla, se ha creado un módulo

`node_link`, que se encarga simplemente de recibir dos interfaces de tipo `node_port` y conectarlas combinacionalmente.

Instanciando estos módulos enlace cada vez que queramos conectar dos puertos, hacemos que si cambiamos el interfaz `node_port` solo haga falta cambiar el código de las conexiones una única vez en el módulo `node_link`, sin ser necesario modificar el código de la malla.

6.1.5. Simulación y validación

Para comprobar la resiliencia de la red ante diversos estímulos, en lugar de usar señales generadas a mano con inspección manual de las salidas, se ha creado un testbench con auto-comprobaciones.

Las pruebas consisten en la generación de una serie de *Paquetes* de manera aleatoria (fuente, destino y datos) a lo largo del tiempo que serán encolados y enviados por las distintas entradas de la red, comprobando que, eventualmente, llegan a su destino. La idea es simular una serie de computadores que generan paquetes de tamaño aleatorio (siguiendo una distribución uniforme), con una probabilidad p de que cada dispositivo genere un paquete en un momento dado.

Para simular distintas condiciones de red, la probabilidad p va cambiando a lo largo de la simulación, aunque es la misma para todos los dispositivos. Durante los primeros 30000 ciclos de reloj, la probabilidad de que un dispositivo genere un paquete es fija y es de 0,01 %. Nótese que cada ciclo de reloj se calcula esa probabilidad para cada uno de los emisores, por lo que la probabilidad de que en un ciclo se genere un paquete es mayor, y para calcularla hay que tener en cuenta las dimensiones de la red. A partir de los 30000 ciclos, p se aumenta linealmente a un ritmo de 0,01 % cada 100, hasta llegar a la saturación de la red (cuando la cola del generador está llena). Momento en el que se dejan de enviar paquetes y, eventualmente, debería vaciarse la red.

El campo de pruebas cuenta con los siguientes procesos concurrentes (visualizados en la figura 6.4):

- Un **generador** de paquetes que, por cada ciclo de reloj, genera distintos paquetes con su fuente y destino con probabilidad p, y los envía a la cola de los emisores y los receptores correspondientes.
- Tantos **emisores** como puertos de entrada tiene la NoC. Van consumiendo paquetes de la cola, troceándolos en flits, y enviandolos por la red respetando el protocolo de transmisión. También es capaz de detectar bloqueos contando el número de ciclos necesarios para establecer una conexión. Si ese número de ciclos es mayor que una constante, lanza un *warning*.

- Tantos **receptores** como puertos de salida tiene la malla. Comprueban que los paquetes que le llegan son para ese destino, y que han sido emitidos previamente. Para ello, tienen una lista con los flits generados por el generador con este destino. Al terminar el test, si todavía quedan paquetes en dicha lista, es porque han sido emitidos pero no recibidos (perdidos), por lo que se imprimen y el test acaba con errores.

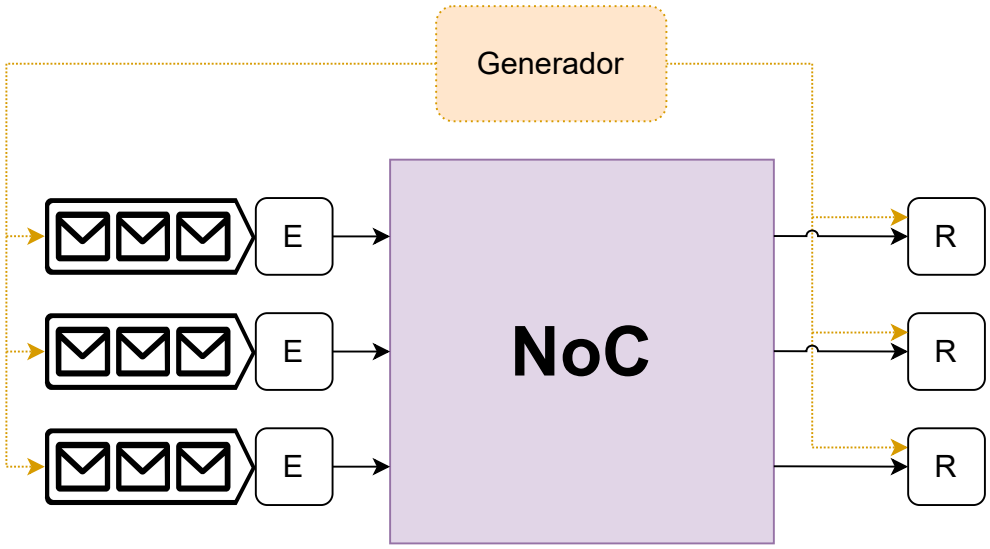


Figura 6.4.: Representación del funcionamiento del *testbench*. Existen tantos emisores (E) como receptores (R), uno por cada puerto de entrada o salida de la red.

La ejecución de la simulación genera una serie de ficheros en formato de tabla separado por comas (CSV) que pueden ser procesados con herramientas externas para obtener más información del estado de la red durante la simulación. La tabla 6.3 muestra los datos de los paquetes incluidos en los CSV. Los resultados de la simulación y el análisis de los datos generados se presentan en la sección 8.1

6.1.6. Envoltorio post-síntesis

La síntesis con Vivado tiene ciertas restricciones. Para hacer que los módulos sean compatibles con herramientas que reciben *stubs* de Verilog, como este no soporta arrays ni interfaces, el sintetizador modifica el nombre de las señales de entrada/salida de nuestros módulos para hacerlos compatibles. En consecuencia, se "aplanan" los arrays, generando una señal por cada uno de los elementos de nuestro array. Como efecto secundario, esto hace que nuestro *testbench*, que era válido para pruebas de comportamiento, ya no sea compatible con el modelo sintetizado debido al cambio en el nombre de las señales.

Para solucionar este problema se ha creado un envoltorio (*wrapper*) que, dependiendo de

Nombre	Descripción
p_id	identificador del paquete
gen_cycle	ciclo en el que se genera el paquete
src_x	ordenada x de la fuente del paquete
src_y	ordenada y de la fuente del paquete
dst_x	ordenada x del destino del paquete
dst_y	ordenada y del destino del paquete
p_size	Tamaño de los datos del paquete
prob	La probabilidad de generar un paquete

(a) Columnas del CSV creado por el proceso *generador*.

Nombre	Descripción
p_id	identificador del paquete
hdr_cycle	ciclo en el que comienza la emisión de la cabecera
sending_cycle	ciclo en el que comienza la emisión de datos
tail_sent_cycle	ciclo en el que finaliza la emisión de datos

(b) Columnas del CSV escrito por los procesos *emisores*.

Tabla 6.3.: Columnas de los ficheros CSV *packets.csv* y *senders.csv*, modificados por el *generador* y los *emisores* respectivamente.

una constante de compilación, utiliza el nombre de las señales originales (para simulación pre-síntesis), o "convierte" el nombre de las señales usando *assigns* y *wires*. Debido a que es necesario generar tantas señales nuevas como elementos hubiese en los arrays de los puertos de la NoC, se ha creado un pequeño *script* en Python que genera parte de este envoltorio. Sin embargo, como el wrapper recibe el tamaño de la NoC por parámetro para pasárselo a la malla, el código generado por el script contiene condicionales de tiempo de compilación para evitar tener que regenerar el wrapper usando el script cada vez que cambiemos el tamaño de la red.

6.2. Integración de la NoC en el SweRV-EL2

La NoC se usa dentro del RISC-V para conectar distintos elementos de la unidad de ejecución (EXU). En su configuración original, estas conexiones son buses de señales punto a punto. Por lo tanto, ha sido necesario generar módulos que conviertan estas señales a paquetes, y los transmitan/reciban por la red siguiendo el protocolo especificado en la sección 4.5.2.

6.2.1. Emisor

El módulo `sender` se encarga de recibir un bus de longitud preestablecida, empaquetarlo en flits y enviarlo por la red a un receptor. Para disminuir la cantidad de flits a enviar, es posible enviar una cantidad reducida de bits en el espacio libre de la cabecera.

El módulo tiene dos parámetros: **PACKET_BITS** que establece la longitud de los bits a ser enviados en el paquete y **PADDING_BITS** que establece la longitud de los bits a ser enviados en la cabecera, y debe ser menor que la cantidad de bits en la parte libre de la cabecera. Automáticamente se calcula un parámetro local con el número de flits a enviar, dividiendo *PACKET_BITS* por la cantidad de bits por flit.

Para enviar un paquete, se habilita la señal *enable*, comenzando la emisión de los datos de los buses de entrada. Una vez se han enviado todos los flits que forman un paquete, se pone a 1 la señal de salida *ack*. Si queremos enviar otro paquete primero es necesario establecer a 1 la entrada *flush* para indicar que los datos de entrada han cambiado y puede volverse a emitir el mensaje. Los puertos de entrada y salida del módulo quedan definidos en la tabla 6.4.

Para mantener el diseño lo más sencillo posible y minimizar la cantidad de registros necesarios, es imprescindible que las entradas permanezcan estables durante la emisión del paquete, pues no se guardan en registros.

Dir.	Tipo	Nombre	Descripción
I	logic	clk	El reloj de la red
I	logic	rst	Reset asíncrono para los registros de la FSM
I	logic	enable	Podemos comenzar la emisión
I	logic	flush	Queremos enviar nuevos datos de entrada
I	logic	dst_addr	La dirección a la que enviar los datos
I	[PADDING_BITS]	padding	Los bits a embeber en la cabecera
I	[PACKET_BITS]	packet	Los bits a empaquetar y enviar tras la cabecera
O	logic	ack	Si se ha conseguido enviar el paquete
O	node_port	up	El puerto de salida a conectar con la NoC

Tabla 6.4.: Puertos de un *serial sender*.

La figura 6.5 presenta el diagrama de transición de estados de la FSM del emisor, que cuenta con los siguientes estados:

- ***IDLE***: No se están enviando datos. Si se activa *enable*, pasa a *ESTABLISHING* y comienza el envío de la cabecera.
- ***ESTABLISHING***: Se establece el camino. Para ello, se reenvía la cabecera hasta recibir el *ack* de la conexión, con lo que pasa al estado *SENDING*.

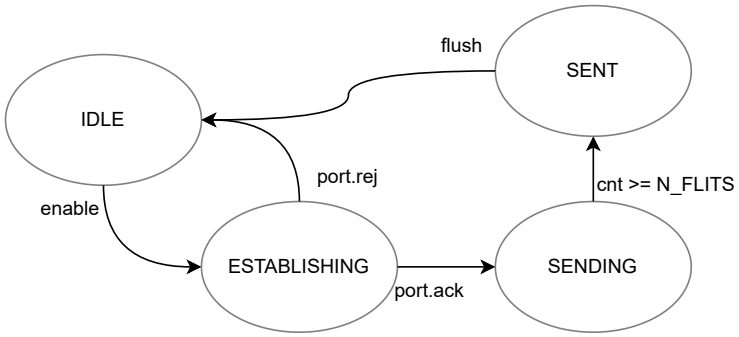


Figura 6.5.: FSM del emisor de datos recibidos en serie.

- ***SENDING***: Se envían el resto de flits que forman el paquete. La cantidad de flits *N_FLITS* es constante, y es simplemente el número de bits a enviar dividido por los bits que caben en cada flit. Nos mantenemos en este estado hasta que hayamos enviado todos los flits, tras lo cual pasa al estado *SENT*.

- ***SENT***: La emisión queda bloqueada hasta que se reciba la señal *flush*, enviada por el usuario del emisor, haciendo que el emisor sea *one-shot*.

6.2.2. Receptor

El receptor tiene un diseño similar al emisor, si bien su tarea es precisamente la contraria. Se encarga de recibir un paquete, dividido en flits, que ha sido enviado por un emisor y desplegarlo en un bus. Tiene, por lo tanto, los mismos parámetros que el emisor (véase 6.2.1) y un funcionamiento similar. La tabla 6.5 muestra los puertos de entrada y salida de este módulo.

Cuando el receptor ha terminado de recibir un paquete y este está disponible en los puertos *padding* y *packet*, pone la señal de salida *valid* a 1. Una vez se haya consumido este dato, el módulo usuario debe poner la señal de *flush* a 1 para que se pueda recibir otro paquete.

Dir.	Tipo	Nombre	Descripción
I	logic	clk	Reloj de la red
I	logic	rst	Reset asíncrono para los registros de la FSM
I	logic	flush	Liberar los recursos
I	node_port	down	Puerto de entrada a conectar con la NoC
O	logic	valid	Si *padding* y *packet* son válidos
O	[PADDING_BITS]	padding	Los bits embebidos en la cabecera
O	[PACKET_BITS]	packet	Los bits recibidos en el cuerpo de los flits

Tabla 6.5.: Puertos de un *serial receiver*.

La figura 6.6 presenta la FSM del `receiver`, que cuenta con un estado menos que el `sender`:

- ***IDLE***: Los recursos están disponibles y se puede recibir información. Con la salida *down.ack* a 1 se señaliza al *router* conectado que es posible recibir datos. Al recibir una cabecera, se almacenan en un registro los *PADDING_BITS*, pasando al estado *RECEIVING*.

- ***RECEIVING***: Debido a la conmutación de circuitos segmentada es posible recibir la cabecera repetida, por lo que es necesario ignorarla hasta recibir el primer flit de datos. Después, se almacena la carga de los *flits* de datos recibidos en el registro correspondiente hasta recibir un flit de tipo TAIL, pasando al estado *VALID*. Para saber qué registro habilitar, se usa un contador del número de flits de datos recibidos, que será la entrada de un decodificador cuya salida serán las señales *enable* de cada uno de los registros que almacenan la salida *packet*. Es decir, aunque todos los registros reciben la misma entrada (los datos del flit), solo se escribe en el registro necesario.

- ***VALID***: La información del paquete está disponible en las salidas *padding* y *packet*. Se espera a que la salida sea consumida hasta que el usuario del módulo habilite la señal de *flush* para poder recibir paquetes de nuevo. Durante este estado la salida *down.ack* permanece desactivada para que el nodo adyacente no establezca conexiones con este dispositivo, evitando que se pierdan los paquetes o se sobreescriban los registros.

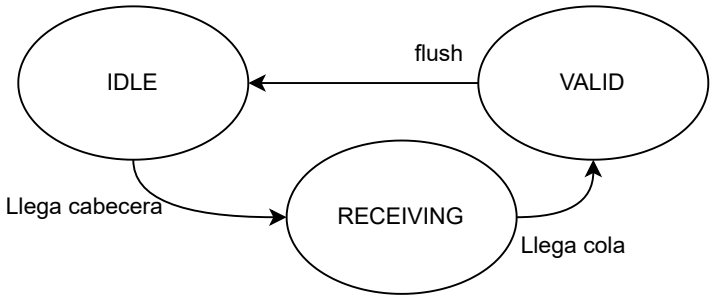


Figura 6.6.: Diagrama de la FSM del receptor y desempaquetador de datos.

En conjunto, el emisor y el receptor permiten enviar datos flit a flit a través de la NoC. En la figura 6.7 se muestra el diagrama de señales de una emisión de 42 bits de datos (3 flits y cabecera) a través de una red pasando por tres nodos. Debido a la conmutación de circuitos segmentada, como el paquete pasa por tres nodos, es necesario que el emisor replique la cabecera tres veces hasta recibir el *ack*, permaneciendo en el estado *ESTABLISHING*. El receptor pasa al estado *RECEIVING* el mismo ciclo de reloj que el emisor pasa al estado *SENDING*, sin embargo, el receptor debe descartar la cabecera durante los 3 primeros ciclos. A continuación permanece en dicho estado mientras recibe los 42 bits del paquete (3 flits).

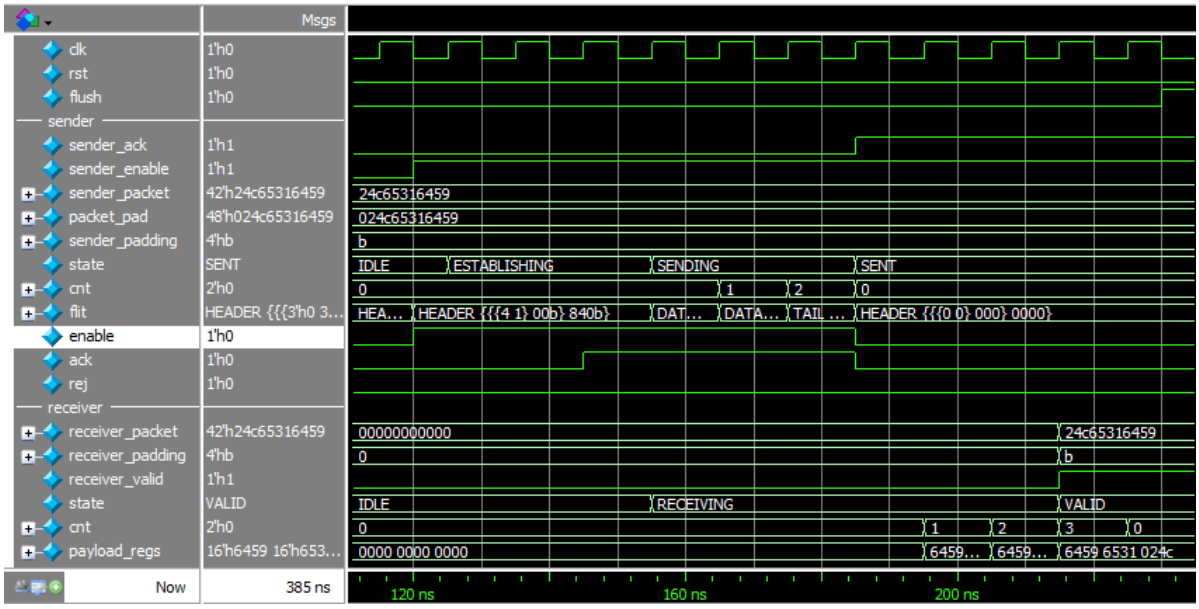


Figura 6.7.: Señales de entrada y salida de un banco de pruebas con un emisor y un receptor, enviando un paquete de 3 flits a una distancia de 3 nodos. Nótese como los datos enviados en la señal *sender_packet* acaban siendo replicados en la salida *receiver_packet*.

Puede observarse el momento en el que el receptor deja de descartar cabeceras y empieza a consumir flits de datos observando cuando aumenta la señal *cnt* del *receiver*. Finalmente, tras recibir los tres flits de datos del paquete se pasa al estado *VALID*, desplegando los bits en la señal *receiver_packet* (que es igual a *sender_packet*), y habilitando *receiver_valid*.

6.2.3. Wrappers en la unidad de ejecución

Se han creado distintos *wrappers* que encapsulan cada uno un elemento de la unidad de ejecución. De esta forma, se crea un módulo que se conecta únicamente con la red y que mantiene el mismo comportamiento que el elemento al que envuelve.

Usando el receptor y el emisor mencionados en las secciones anteriores podemos convertir los paquetes de la red a señales del módulo a envolver, y los resultados de ese módulo de nuevo a paquetes.

En la figura 6.8, el receptor `i_receiver` desempaqueta los flits recibidos por el puerto *down.flit*, y los pasa a las entradas *dividendo* y *divisor* del módulo `divisor`. Cuando el divisor termina de realizar los cálculos, se envía el resultado (*out*) al emisor `i_sender`, que lo empaqueta y lo manda de nuevo por la red.

Para que el envoltorio del divisor pueda recibir y enviar datos a las otras unidades del procesador, es necesario un emisor y receptor encargados de transformar estas señales y mandárselas al wrapper, como podemos ver en la figura 6.9.

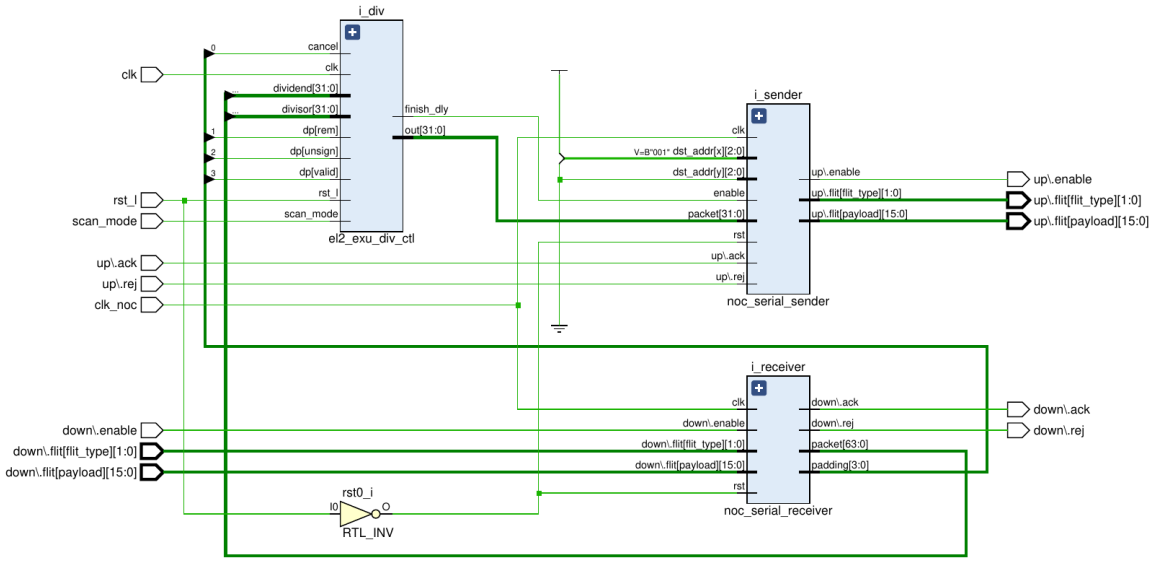


Figura 6.8.: Diagrama de bloques del módulo *div_wrapper*, envoltorio del divisor.

6.2.4. Modificaciones en la unidad de ejecución

La unidad de ejecución del SweRV-EL2 (EXU o *Execution Unit*) se encarga principalmente de la fase de ejecución de las instrucciones, por lo que recibe los datos con los que operar (extraídos por las fases de *fetch* y *decode*) y devuelve el resultado de la operación (que será usado en la fase de *retire*). Asimismo, la EXU se encarga también de otras operaciones no explicitadas en una instrucción, como predicciones de salto o cálculos relativos al contador del programa.

Para ello, la unidad de ejecución instancia los siguientes módulos:

- **i_alu**: Es la Unidad Aritmético Lógica (ALU) y se encarga de hacer operaciones sencillas en menos de un ciclo de reloj, como pueden ser sumas y restas. Es usada para la ejecución de instrucciones aritmético-lógicas, pero también para operaciones con el contador de programa y de predicción de salto.

- **i_mul**: Se encarga únicamente de realizar multiplicaciones, implementando la extensión *M* del ISA. El resultado está disponible en el siguiente ciclo.

- **i_div**: A diferencia de los dos anteriores, este dispositivo se encuentra fuera del

pipeline principal (decimos que es *out-of-pipe*) y tiene 34 ciclos de latencia. No está segmentado ni tiene cortocircuitos dentro de la ALU.

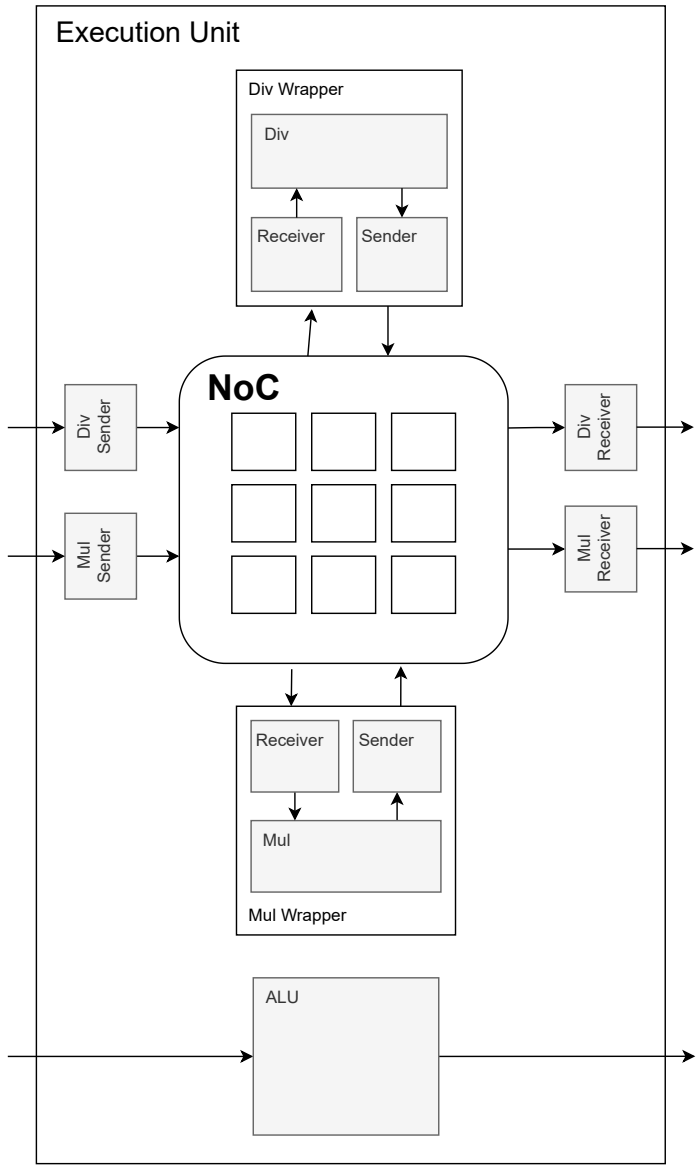


Figura 6.9.: Diagrama de bloques de la unidad de ejecución y los módulos necesarios para el funcionamiento del multiplicador y divisor.

En este proyecto se ha añadido una NoC a la unidad de ejecución. Debido a que la NoC (definida en el capítulo 4) necesita de varios ciclos de reloj para enviar un paquete, ha sido necesario añadir un nuevo reloj interno *clk_noc* que recibe cada uno de los elementos que conforman la NoC, incluidos los emisores y receptores. En las pruebas realizadas, dicho reloj va 16 veces más rápido que el reloj del sistema.

Además, se ha instanciado una malla de tamaño 3×3 (configurable con definiciones del precompilador), y se han instanciado los emisores, receptores y *wrappers* correspondientes alrededor de la NoC para el multiplicador y el divisor, creando paquetes con las siguientes características:

- **Entrada MUL**: Recibe los dos operandos de 32 bits, y un estructurado de 23 bits con información sobre la operación a realizar (si es válida, el signo, etc...). Por lo tanto, el paquete generado tiene $\lceil \frac{32+32+23}{FLIT\_DATA\_WIDTH} \rceil = 6$ flits de datos a 16 bits cada uno.

- **Salida MUL**: Se envía un resultado de 32 bits, por lo que tendremos 2 flits de datos. Dentro del *wrapper*, para evitar enviar siempre el resultado (aunque no se haya hecho ningún cómputo), como el multiplicador no expone ninguna señal de valid, activamos el *enable* del emisor un ciclo después de que entrasen datos al multiplicador, evitando así la sobrecarga de la red.

- **Entrada DIV**: Se envía el numerador y el denominador (de 32 bits), y un estructurado de 3 bits con información de la operación (el signo, y si queremos el cociente o el resto). Como dicho estructurado tiene solo 3 bits, podemos enviarlo en los bits libres de la cabecera, y hacemos que se necesiten tan solo 4 flits de datos para enviar entre el emisor y el *wrapper*.

- **Salida DIV**: Se envía el resultado y una señal que indica que la división ha acabado (pues el divisor es *out-of-pipe*). En lugar de mandar esta señal en el paquete, la usamos para activar el emisor del *wrapper*, y es el receptor el que se encarga de activarla al recibirlo (usando la señal *valid* del módulo *serial_sender*). Por lo tanto, este resultado necesita de 2 flits de datos para enviarse.

Es decir, **el tamaño máximo de un paquete que pasará por nuestra red es de 7 flits** contando la cabecera, siendo datos para una multiplicación.

En cuanto a la ALU, se ha decidido no incluirla en la NoC debido a su complejo funcionamiento por sus múltiples responsabilidades. Sería necesario realizar numerosas modificaciones para adaptarla, expuestas en el capítulo 10.

Todos estos cambios han sido parametrizados con una definición del precompilador `` `RV_EXU_NOC ``, que nos permite decidir en tiempo de compilación si queremos o no que la EXU utilice la NoC sin ser necesario tener dos proyectos distintos.

Detector de cambios en reloj

Como hemos explicado en sus respectivas secciones, el emisor y el receptor son *one-shot*, es decir, tras realizar su acción se bloquean hasta recibir una señal de desbloqueo llamada *flush*.

En el caso del RISC-V con el objetivo de mandar las nuevas señales que pueden haberse generado este ciclo de reloj es necesario mandar la señal de *flush* cuando estas cambien.

Para ello se ha creado un detector de cambio de reloj. Este módulo recibe el reloj del que queremos detectar el *rising edge* y un reloj base en el que apoyarnos y que alimentará la lógica secuencial del detector. El módulo tendrá como salida una señal *rising* que se pondrá a uno cuando se detecte un cambio de $0 \rightarrow 1$ en el reloj principal.

La idea principal es guardar en un registro el estado de la señal el ciclo anterior, y compararla con el estado actual para comprobar si se ha producido un cambio. Esta comparación es simplemente la conjunción lógica, por lo que la salida de nuestro módulo será dada por la fórmula $rising = clk\_slow \ \& \ q$, donde q es el registro con la señal desfasada.

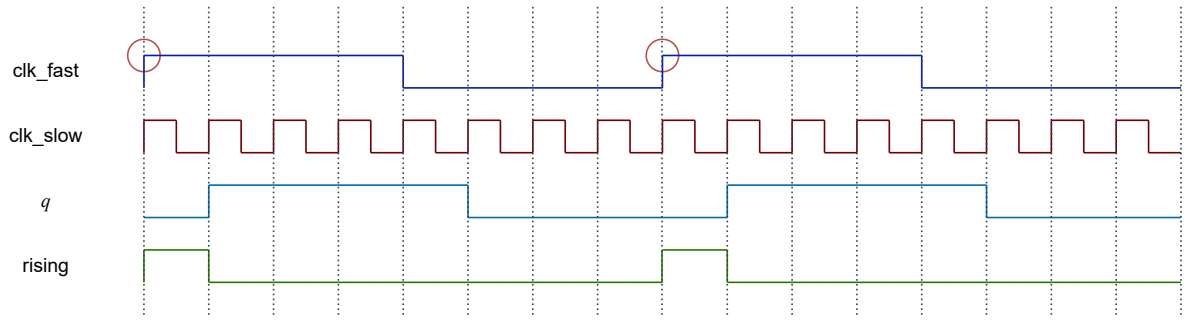


Figura 6.10.: Diagrama temporal de las señales de un detector de cambios de reloj, con un reloj del que detectar el cambio (*clk_fast*) 4 veces más lento que el base (*clk_slow*). También se visualiza la señal interna q.

Capítulo 7.

Problemas y obstáculos encontrados

> «Some problems are easy to find and hard to fix; some are hard to find and easy to fix; some go both ways.»
> – Tracy Kidder, *The Soul of a New Machine*

En esta sección se presentan los distintos problemas y obstáculos encontrados durante la realización del proyecto. Tareas que han ocupado tiempo pero, al no dar resultados, no han sido mencionadas en otras partes de la memoria.

7.1. Diseño de una NoC asíncrona

Al comienzo del proyecto se decidió que la conmutación de circuitos de la NoC fuese asíncrona, pudiendo establecer un camino en un ciclo de reloj y, en el siguiente ciclo, comenzar el envío de flits de datos. Esto no debería causar ningún problema gracias a que el algoritmo de encaminamiento, *Dimensional Order Routing*, está libre de *deadlocks*.

Además, la simulación *conductual* se ejecutaba sin errores. Sin embargo, al hacer simulaciones post-síntesis (teniendo en cuenta el retardo de las puertas), la simulación se atascaba con algunas señales inestables.

El problema principal era que se podía reenviar una cabecera por el puerto incorrecto durante unos pocos picosegundos, pues los datos se propagaban más rápido que las señales de control del `crossbar`. Esa cabecera podía, a su vez, ser reenviada de nuevo creando una situación inestable en la que aparecía un ciclo combinacional. Estos problemas ocurrían al realizar una síntesis sin restricciones, y se podrían solucionar añadiendo *constraints* a las señales, restringiendo las entradas del `crossbar` para que la señal con los nuevos datos llegue **después** de haber realizado la conmutación de circuitos. Estas tareas, sin embargo, no son a nivel de diseño RTL, por lo que se buscaron alternativas a dicho nivel para lograr solucionar el problema. Tras múltiples semanas intentándolo, se determinó abandonar dicho diseño y cambiar las especificaciones de la red, buscando

un diseño síncrono para eliminar dichos ciclos combinacionales.

Finalmente, se decidió pasar a un diseño con establecimiento de caminos secuencial, pero intentando reutilizar la mayor parte posible del diseño anterior, por lo que se implementó el algoritmo de conmutación de circuitos segmentado.

Puede verse el antiguo diseño con conmutación de caminos combinacional usando el sistema de control de versiones git, en el repositorio en GitHub del proyecto[1], mostrando el estado con un *commit* anterior al 3 de Mayo de 2022.

7.2. Problemas al conectar la ALU y la NoC

La unidad aritmético-lógica (ALU) ha sido el único módulo de la unidad de ejecución que no utiliza la NoC. Esto es debido principalmente a los siguientes problemas:

1. Tiene muchas responsabilidades, no solo se usa en la fase de ejecución de las instrucciones aritmético-lógicas, sino que también se utiliza en la predicción de salto o en el cálculo de direcciones de memoria en *load* y *store*.

2. Por lo mencionado en el ítem anterior, la ALU cuenta con demasiadas señales, haciendo que los paquetes tengan demasiado tamaño y tarden demasiado en llegar. Por lo tanto, sería necesario aumentar la frecuencia de reloj de la NoC para que llegasen en menos de un ciclo de reloj del sistema.

3. Además, la ALU no expone cuando se ha terminado de realizar un cálculo, por lo que el *wrapper* no puede identificar cuando deben enviarse los resultados.

En la sección 10.2.1 se mencionan posibles soluciones a dichos problemas.

[1] `https://github.com/daviddavo/tfg_poc_noc`

Parte IV.

Conclusiones y resultados

Capítulo 8.

Resultados de las simulaciones conductuales

«It works, it works! I finally invented something that works!»
– C. Lloyd como Dr. Emmet Brown en *Regreso al Futuro*

8.1. Simulación conductual de la NoC

En esta sección se analizan los datos generados por la simulación de comportamiento del RTL de la NoC de 3×3 y 16 bits por flit, explicada en la subsección 6.1.5. Para ello se han usado herramientas de análisis de datos como Pandas y Numpy, generando las gráficas con Matplotlib. En la carpeta `data` del repositorio del proyecto[1] pueden encontrarse los datos generados por la simulación y el Jupyter Notebook para el procesado de estos.

Durante la prueba de la red se emiten paquetes con información generada aleatoriamente, de un tamaño también aleatorio de entre 1 y 160 bits (2 a 12 flits, contando la cabecera) siguiendo en ambos casos distribuciones uniformes. Por lo tanto, el tamaño medio de los paquetes es de 6,5 flits. Para simular distintas condiciones de red, la cantidad de paquetes en la NoC aumenta de manera lineal conforme avanza el tiempo de la simulación.

Las simulaciones, tanto conductuales como post-síntesis (teniendo en cuenta los retardos de las puertas y conexiones), han sido satisfactorias, sin producirse ningún tipo de bloqueo (véase 4.2.1 Problemas de encaminamiento), ni pérdida de paquetes.

En total, en la NoC se han emitido 31009 paquetes a lo largo de más de 100000 ciclos, enviando un total de 2.5 megabits de datos haciendo uso de 171997 flits.

[1] `https://github.com/ogarnica/TFG2021-22_RISC-V_NoC/tree/main/data`

8.1.1. Detección de problemas

La probabilidad de que se presenten problemas en una red aumenta cuanto mayor sea la carga de la red, pues más recursos estarán bloqueados al mismo tiempo y más probable es que se intente acceder a un recurso bloqueado. Por ello, se han hecho pruebas aumentando la carga de la red de manera lineal. El único problema detectado en nuestra red es cierto grado de *starvation* cuando la carga es muy elevada, pero nunca superando los 16 ciclos de espera. La figura 8.1 muestra que hay una fuerte correlación entre la ocupación de la red y el tiempo de espera, llegando a su máximo a partir del ciclo 80000, en el que hay 11 paquetes haciendo uso de la red y el tiempo de espera medio es de 15 ciclos. Para calcular el número de paquetes en la red se han restado la suma cumulativa del número de paquetes salientes y la suma cumulativa del número de paquetes que han establecido una conexión, por lo que no se cuentan las emisiones en las que aún se está estableciendo una conexión, a pesar de que también bloquean recursos.

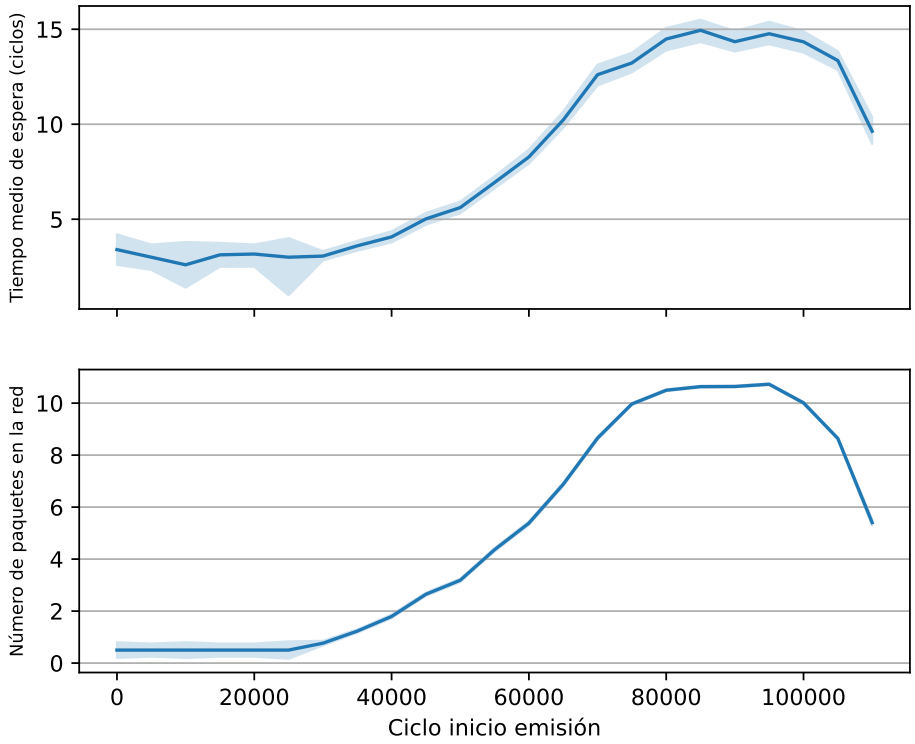


Figura 8.1.: Tiempo de espera medio hasta inicio de emisión y carga de la red en una red 3x3 (el sombreado indica la desviación típica de la media). La carga de la red aumenta desde el ciclo 30000 de manera linear hasta saturar la red.

8.2. Simulación conductual del SweRV-EL2

Para verificar el procesador, ha de utilizarse un simulador para ejecutar un modelo del diseño RTL, cargando en la memoria del procesador cualquier programa escrito en C o ensamblador.

Con objeto de facilitar esta tarea, una de las herramientas incluidas en el proyecto original del SweRV-EL2 es un fichero *Makefile*, que compila el programa a cargar en la memoria y simula el *core*, permitiendo seleccionar el simulador a utilizar.

Se ha modificado la parte del *Makefile* para que, cuando se seleccione *QuestaSim* como simulador, se incluyan también los ficheros de la NoC. También ha sido necesario modificar el *testbench* (fichero SystemVerilog encargado de definir los estímulos de entrada y verificar las salidas), añadiendo únicamente el nuevo reloj para la NoC de la EXU.

Durante el desarrollo, para comprobar el funcionamiento del procesador tras realizar pequeñas modificaciones en el diseño, se ha ejecutado un pequeño programa que imprime las palabras "Hello World" en la salida estándar del simulador (figura 8.2). Si dicho test termina satisfactoriamente, se procede a ejecutar *CoreMark*, un *benchmark* diseñado para probar la funcionalidad de un core [11], ejecutando todo tipo de instrucciones. En la figura 8.3 se presentan los resultados de dicho benchmark.

Observando los registros de ejecución generados en el fichero *exec.log*, puede comprobarse que se ejecutan instrucciones de multiplicación y de división sin problemas, por lo que la implementación de la NoC ha sido satisfactoria.

```
# run -all
# ---------------------------------
# Hello World from SweRV EL2 @WDC !!
# ---------------------------------
# TEST_PASSED
#
# Finished : minstret = 437, mcycle = 922
# See "exec.log" for execution trace with register updates..
#
# ** Note: $finish    : E:/TFG/TFG2021-22_RISC-V_NoC/Cores-SweRV-EL2/testbench/tb_top.sv(343)
#    Time: 9280 ns  Iteration: 1  Instance: /tb_top
# End time: 16:49:55 on May 29,2022, Elapsed time: 0:00:13
# Errors: 0, Warnings: 1375
```

Figura 8.2.: Resultados de la simulación del programa *hello_world*.

```
# run -all
# 2K performance run parameters for coremark.
# CoreMark Size    : 666
# Total ticks      : 416012
# Total time (secs): 416
# Iterat/Sec/MHz   : 2.57
# Iterations       : 1
# Compiler version : GCC9.2.0
# Compiler flags   : -g -O3 -funroll-all-loops
# Memory location  : STATIC
# seedcrc          : 0xe9f5
# [0]crclist       : 0xe714
# [0]crcmatrix     : 0x1fd7
# [0]crcstate      : 0x8e3a
# [0]crcfinal      : 0xe714
# Correct operation validated. See readme.txt for run and reporting rules.
# TEST_PASSED
#
# Finished : minstret = 304107, mcycle = 440127
# See "exec.log" for execution trace with register updates..
#
# ** Note: $finish    : E:/TFG/TFG2021-22_RISC-V_NoC/Cores-SweRV-EL2/testbench/tb_top.sv(343)
#    Time: 4401330 ns  Iteration: 1  Instance: /tb_top
# End time: 16:44:17 on May 29,2022, Elapsed time: 0:01:55
# Errors: 0, Warnings: 1375
```

Figura 8.3.: Resultados de la simulación del programa *Coremark*.

Capítulo 9.

Resultados de la síntesis del SweRV-EL2 y NoC

> «It's one thing when you plug in to a socket in the wall and electrons flow. It's another thing when you have to figure out, for every electron, which direction it takes.»
> – Bob Kahn en *Where Wizards Stay Up Late*

Aunque el objetivo principal del proyecto es realizar un diseño RTL, se ha sintetizado dicho diseño para la plataforma objetivo Virtex-7 VC707 Evaluation Kit, que hace uso de una FPGA Virtex-7 XC7VX485T-2FFG1761C [1].

Esta FPGA cuenta con 700 pines de entrada/salida, más de 300 mil LUTs y 600 mil registros.

9.1. Área utilizada por el diseño

Se ha generado un reporte de utilización usando Xilinx Vivado tras sintetizar para la Virtex-7 con la configuración por defecto. Debido a que aún no se ha realizado la fase de implementación ni sus optimizaciones, estos resultados son una mera estimación, pudiendo ser diferentes al realizar el flujo de implementación a la FPGA.

En total, el diseño sintetizado (que incluye *core*, memoria e interfaz DMI) ocupa un 14 % de las LUTs de la FPGA (40 mil), y un 4 % de sus registros (23 mil), siendo con diferencia el *core* el que más área consume: un 12 % y 3,59 % de los recursos disponibles en la FPGA respectivamente (36 mil y 21 mil).

Dentro del *core*, su elemento más pesado es la IFU, con 11 mil LUTs (3,69 %), seguido de la EXU con 10 mil LUTs (3,18 %). Por lo tanto, solo estos dos elementos ya forman el 40 % de los LUTs consumidos por el core, dejando un 3,77 % del total (11 mil LUTs)

para el DEC y el LSU. Por otra parte, dos tercios de los registros del *core* son utilizados por el IFU. En la figura 9.1 se visualizan dichos áreas. Todos los *senders* y *receivers* instanciados a este nivel (es decir, sin contar los de dentro de los wrappers) sumados hacen uso de 66 LUTs y 79 flip-flops, siendo los receptores los que más registros consumen (70 de los 79), puesto que deben almacenar los datos del paquete recibido. El detector de reloj ha sido sintetizado a un flip-flop.

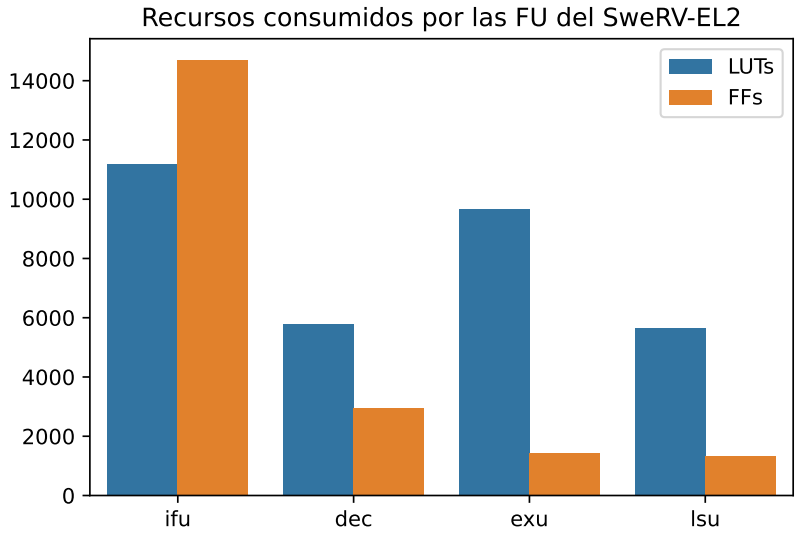


Figura 9.1.: Recursos consumidos por las unidades funcionales del *core*.

9.1.1. Recursos utilizados en la unidad de ejecución y la NoC

A continuación analizaremos el uso de recursos de la EXU, a la que se le ha incluido una NoC. En la figura 9.2 se visualizan los datos mencionados a continuación. De los 9700 LUTs usados por la EXU, 2700 (28 %) son ocupados por los encaminadores y conexiones de la NoC. A los wrappers del multiplicador y el divisor se les asignan 1433 y 744 LUTs respectivamente. Otra gran parte de los recursos de la EXU (2214 LUTs, 23 %) son asignados al módulo *i_x_ff*, encargado de parte de la lógica de predicción de salto.

Finalmente, analizaremos los recursos consumidos por los *wrappers*, para discernir si su tamaño es debido a las unidades de cálculo (el multiplicador y el divisor en sí), o por los elementos adicionales (emisores y receptores). Se presentan una visualización de los datos en la figura 9.3. En cuanto al divisor, su *wrapper* ocupa en total 1433 LUTs, y 1368 de estos son para el submódulo divisor, por lo que los elementos adicionales de comunicación de la red emplean tan solo 65 LUTs (un 5 % del wrapper). Por otro lado, el envoltorio del multiplicador ocupa un total de 744 LUTs, mientras que su submódulo

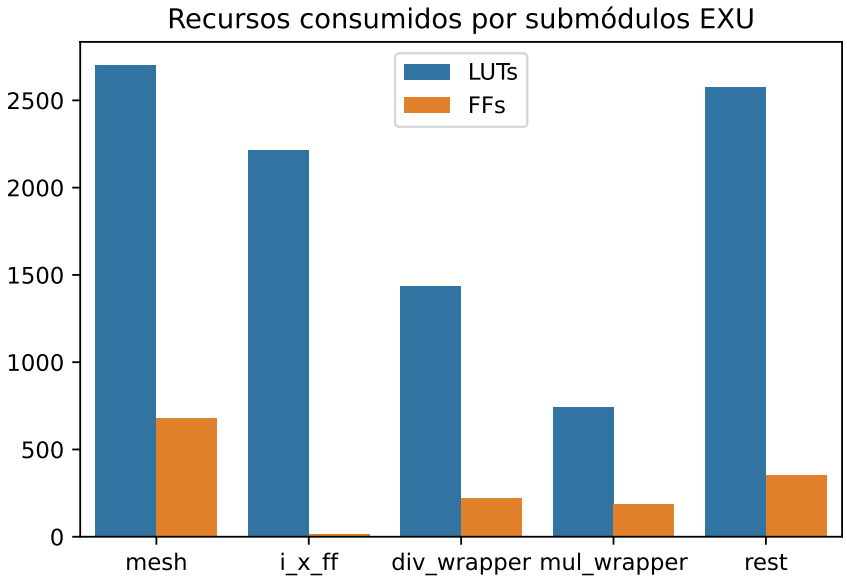


Figura 9.2.: Recursos consumidos por los submódulos de la unidad de ejecución. En rest se muestran los LUTs y *flip-flops* utilizados tanto para el resto de submódulos de la EXU como para la lógica y conexiones del módulo EXU.

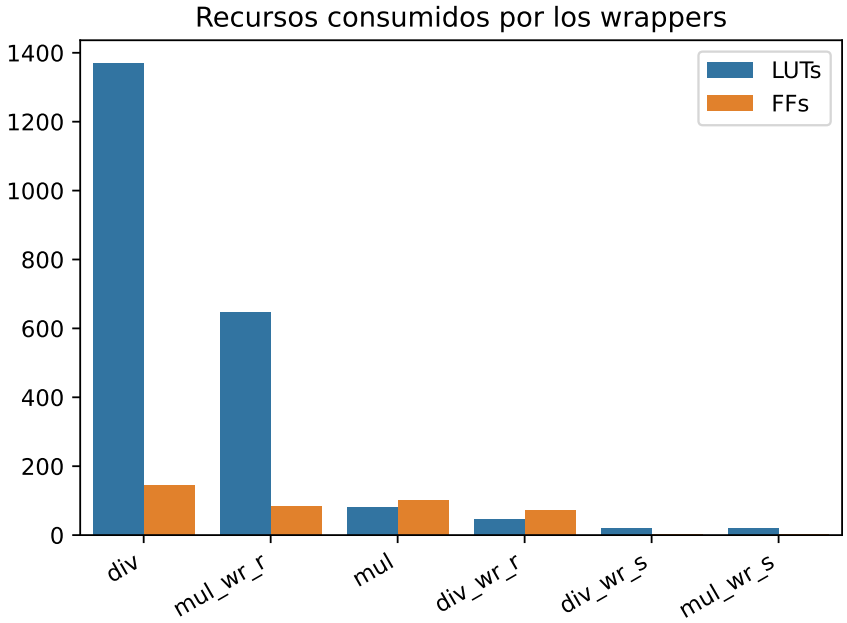


Figura 9.3.: Recursos consumidos por los *wrappers* de los submódulos de la EXU.

principal tan solo cuenta con 79 de estos, dejando que los dispositivos de red utilicen 665 LUTs, el 90 % del wrapper. Esto es debido a que el receptor tiene un gran tamaño, empleando 646 LUTs (86 % del wrapper).

Para concluir, comprobaremos qué parte de los recursos de la EXU son consumidos por los módulos añadidos. Si sumamos los recursos consumidos por todos ellos (malla y emisores/receptores, tanto dentro como fuera de *wrappers*), tenemos que se han añadido 3496LUTs y 918 FFs, principalmente debido a la malla (2700LUTs y 678FFs). Estos elementos son el 36 % de los LUTs y el 63 % de los *flip-flops* de la EXU. Es decir, la inserción de la NoC ha aumentado el número de LUTs consumidos por la EXU en 57 %, y el de registros en 174 %. En conclusión, la NoC tan solo ha aumentado el tamaño total del diseño en un 9 % de LUTs y un 4 % de FFs.

Capítulo 10.

Trabajo futuro

En esta sección se presenta el posible trabajo futuro para continuar con el desarrollo del proyecto. También se exponen soluciones a algunos de los problemas encontrados (capítulo 7).

Para cumplir la motivación principal de este proyecto, debería realizarse una implementación en FPGA del RTL generado y, una vez comprobado su funcionamiento, realizar pruebas de la reconfiguración dinámica.

Sin embargo, al enforcarnos en el diseño RTL, se han dejado de lado algunos aspectos que pueden afectar al diseño implementado. Durante las siguientes secciones se explican tanto mejoras funcionales, como mejoras para lograr una implementación satisfactoria.

10.1. Mejoras en el diseño de la NoC

10.1.1. Mejoras en el *crossbar*

El *crossbar* es el mayor contribuyente al número de LUTs consumidos por un *router*. Además, las señales que pasan por el crossbar tardan demasiado en propagarse y causan violaciones de *timing*, debido a la lógica del *round-robin* y el multiplexor.

Para implementar el *round-robin* de manera diferente (tal vez más eficiente), en lugar de usar un contador que indique el puerto de mayor prioridad, podría usarse un registro de desplazamiento de tantos bits como puertos tiene el *crossbar* en el que la posición del bit que esté a 1 indica el puerto de mayor prioridad. Algunas FPGAs cuentan con dispositivos que pueden implementar registros de desplazamiento de manera muy eficiente (más que un contador), aunque consumiría más registros.

Por otro lado, todos los crossbar almacenan siempre el mismo número en el contador, por lo que podría compartirse dicha lógica si es posible, aunque para ello tal vez no sea necesario modificar el diseño RTL (tan solo las opciones de síntesis).

10.1.2. Emisores de serie a paquetes

Los emisores creados para empaquetar y enviar un bus de bits usan un contador para saber qué parte del bus deben enviar. Sin embargo, ese contador siempre se incrementa en uno y el valor máximo es conocido en tiempo de compilación (el parámetro local *N_FLITS*). Por lo tanto, podría remplazarse dicho contador por un número finito de estados en la FSM.

10.1.3. Estado de la red

Aunque se han realizado pruebas exhaustivas, estas no han tenido en cuenta *soft errors* (SUE) que puedan aparecer, y pueden hacer que la red quede en un estado inestable o de bloqueo. Podría solucionarse si se añadiese algún sistema para monitorizar problemas que surjan en la red, por ejemplo vigilando si una conexión lleva demasiado tiempo establecida o si un camino tarda demasiado en establecerse.

10.1.4. Mejoras en la conmutación de circuitos segmentada

Con la configuración actual, si se intenta establecer una conexión con un dispositivo que está desactivado (estableciendo la señal *ack* a 0 a fin de evitar recibir paquetes), es imposible deshacer el camino y liberar los recursos consumidos durante la conmutación de caminos segmentada (PCS). El último encaminador, que se conecta directamente con este dispositivo, seguirá intentando realizar una conexión indefinidamente, bloqueando todos los recursos reservados para dicho camino. Esto puede hacer que, si hay un error al especificar la dirección destino, los caminos utilizados queden por siempre bloqueados. Por lo tanto, una mejora notable de la NoC sería implementar un sistema de vuelta atrás en el que se deshaga el camino si, al llegar al último encaminador del camino, este no logra conectarse.

10.2. Mejoras SweRV

10.2.1. Conexión de la ALU a la NoC

La Unidad Aritmético-Lógica ha sido el único submódulo de la unidad de ejecución (EXU) que no usa la NoC. Esto es debido, principalmente, a los problemas vistos en la sección 7.2. Para conseguir implementarla podría haber dos enfoques:

1. No modificar la ALU y enviar la información realizando modificaciones externas. Por ejemplo, podría aumentarse la frecuencia de reloj, o utilizarse dos emisores (y dos receptores en el *wrapper*) para enviar la información en paralelo. Esta opción es la más sencilla, pero podría no funcionar y consumiría muchos recursos. Además, permanecería el problema de discernir el momento en el que enviar la información.

2. Estudiar en profundidad la ALU y realizar modificaciones a esta si es necesario. Se comentará más de esta solución a continuación.

Una vez estudiada la ALU, podría crearse un sistema de paquetes con tipos (usando los bits libres de la cabecera para codificar el tipo del paquete), en el que tan sólo se mande la información (y los resultados) necesarios en cada momento. Por ejemplo, si se necesita usar la ALU para la fase de ejecución de una instrucción aritmético-lógica, pueden enviarse los operandos en un paquete de poco más de 64 bits (4 flits), mientras que si se trata del cálculo de la predicción de salto, podría ser un paquete enviando únicamente el paquete con la información para ello. Sería necesario, por lo tanto, crear nuevos emisores y receptores en los que se pueda establecer en tiempo de ejecución el tamaño de los paquetes a enviar.

10.2.2. Soporte para paradas

Con la configuración actual, los paquetes deben tardar menos de un ciclo del sistema en enviarse por la NoC. Por lo tanto, si han de enviarse paquetes demasiado grandes, o alguno de ellos tarda demasiado en transmitirse por la red, los submódulos pueden recibir información incorrecta.

Para solucionar este problema sería necesario hacer que todos los submódulos *esperen* a recibir un paquete para comenzar a realizar el cálculo, siendo necesario en ocasiones modificar ligeramente su diseño para añadir señales de control.

Esta mejora relajaría considerablemente las restricciones temporales del reloj de la NoC, pero podría disminuir el CPI del procesador.

10.2.3. Modificar herramientas de configuración y testing

Para incluir la NoC se ha modificado manualmente la configuración *default* generada por la herramienta *swerv_config_gen*. El uso de la NoC depende de una constante en tiempo de compilación que debe definirse manualmente, por lo que una mejora notable para los usuarios del proyecto sería modificar dicha herramienta incluyendo la opción de habilitar la NoC y modificar sus parámetros: tamaño de la NoC, posición de los elementos, e incluso qué elementos incluir.

Además, sólo se ha comprobado el funcionamiento usando QuestaSim, a pesar de que el SweRV-EL2 original cuenta con scripts para otros simuladores. Para facilitar el uso de nuestro diseño a otros desarrolladores, podrían modificarse dichos scripts para que nuestro proyecto soportase otros conjuntos de herramientas, y no solo Xilinx Vivado y QuestaSim.

Capítulo 11.

Conclusiones

En el proyecto se ha creado una NoC funcional y se ha integrado en un procesador RISC-V SweRV-EL2 para comunicar los módulos de la unidad de ejecución. En la realización del trabajo se han cumplido los siguientes objetivos:

1. He llevado a cabo un estudio en profundidad sobre las redes de interconexión en procesadores y sus distintas características.
2. He aprendido a (i) describir diseño hardware en SystemVerilog y realizar *test-benches* automatizados, (ii) sintetizar en Xilinx Vivado y (iii) simular y depurar usando QuestaSim. Además del manejo del lenguaje de scripting TCL, usado por dichas herramientas.
3. He creado el diseño RTL sintetizable de una NoC, comprobando su funcionamiento siguiendo la metodología *constrained random tests*.
4. Me he familiarizado con la arquitectura RISC-V y, en concreto, con el diseño RTL y las herramientas del procesador SweRV-EL2.
5. He aprendido el manejo del conjunto de herramientas incluidas en la distribución para la configuración, enlazado, compilación y simulación de tests de alto nivel con el procesador SweRV-EL2.
6. He modificado el RTL del procesador SweRV-EL2 para incluir en su unidad de ejecución la NoC desarrollada en el proyecto —usada por el multiplicador y el divisor—, logrando pasar las distintas pruebas incluidas con el diseño.
7. He sintetizado el diseño y obtenido una estimación de los recursos hardware necesarios para su implementación sobre una FPGA Xilinx Virtex-7 XC7VX485T-2FFG1761C, analizando el impacto de la NoC.

En conclusión, el nuevo diseño del SweRV-EL2 incluye una NoC en su unidad de ejecución, con un sobrecoste en recursos hardware del 9 % en LUTs y 4 % en FFs con respecto al diseño original. No obstante, la adición de la NoC permite usar técnicas de reconfiguración dinámica parcial cuando se detecten daños permanentes sobre parte del circuito, aplicando técnicas de tolerancia a fallos que hacen posible usar este diseño en entornos especialmente agresivos con los dispositivos electrónicos.

Chapter 11.

Conclusions

In this project, I have created a working NoC and integrated it in the RISC-V SweRV-EL2 processor to connect the submodules of its execution unit. The following objectives have been fulfilled:

1. I have studied interconnection networks and their characteristics in-depth.
2. I have learnt (i) to design hardware using SystemVerilog and automatized test-benches, (ii) to synthesize with Xilinx Vivado, and (iii) to simulate and debug using *QuestaSim*. I also have acquired the handling of the TCL scripting language used by these tools.
3. I have created a synthesizable design of an NoC and checked its behaviour using the constrained random tests methodology.
4. I have become familiar with the RISC-V architecture and, in particular, with the RTL design and tools of the SweRV-EL2 processor.
5. I have learnt to utilize the toolset included in the repository to configure, link, compile, and run software tests on the SweRV-EL2 processor.
6. I have modified the RTL design of the SweRV-EL2 to include an NoC inside its execution unit —used by its multiplier and divisor—, succeeding in passing the various tests included with the design.
7. I have synthetized the design and obtained an estimation of the hardware resources needed to implement it on a Xilinx Virtex-7 FPGA (XC7VX485T-2FFG1761C), analysing the impact of the NoC.

In conclusion, the new SweRV-EL2 design includes an NoC within its execution unit, with an additional cost in hardware resources of 9% in LUTs and 4% in FFs compared to the original design. Nevertheless, adding the NoC allows using dynamic partial reconfiguration methods to recover processor functionality when permanent damage in the FPGA is detected, improving the design fault-tolerant capabilities and allowing its use in especially harsh environments for electronic devices.

Bibliografía

[1] AMD Xilinx. *Xilinx Virtex-7 FPGA VC707 Evaluation Kit*. URL: https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html#documentation.

[2] Krste Asanović y David A Patterson. *Instruction Sets Should Be Free: The Case For RISC-V*. Inf. téc. 2014. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html.

[3] Krste Asanović Rimas Avizienis Jonathan Bachrach Scott Beamer David Biancolin Christopher Celio Henry Cook Daniel Dabbelt John Hauser Adam Izraelevitz Sagar Karandikar Ben Keller Donggyu Kim John Koenig y col. *The Rocket Chip Generator*. Inf. téc. 2016. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[4] Zvonimir Z Bandic y Robert Golla. *SweRV Cores Roadmap*. Inf. téc. 2019. URL: https://riscv.org/wp-content/uploads/2019/12/12.11-14.20a3-Bandic-WD_SweRV_Cores_Roadmap_v4SCR.pdf.

[5] Christopher Celio, David A. Patterson y Krste Asanović. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor*. Inf. téc. Jun. de 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html.

[6] Tony Chen y David A Patterson. *RISC-V Geneology*. Inf. téc. 2016. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-6.html.

[7] ChipVerify. *ChipVerify | SystemVerilog Interface*. URL: https://www.chipverify.com/systemverilog/systemverilog-interface.

[8] Giovanni De Micheli y Luca Benini. *Networks on Chips. Technology and Tools*. 1st. Elsevier, 2006. ISBN: 978-0-12-370521-1.

[9] Departamento de Arquitectura de Computadores y Automática. «Arquitectura de Computadores. Tema 5. Multiprocesadores y redes de interconexión». En: Universidad Complutense de Madrid, 2020.

[10] José Duato, Sudhakar Yalamanchili y Lionel Ni. *Interconnection Networks: An Engineering Approach*. Revised Printing. Morgan Kaufmann Publishers, 2003. ISBN: 1-55860-852-4.

[11] Embedded Microprocessor Benchmark Consortium. *EEMBC CoreMark Benchmark*. URL: https://www.eembc.org/coremark/.

[12] P T Gaughan y S Yalamanchili. «Pipelined circuit-switching: a fault-tolerant variant of wormhole routing». En: *[1992] Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*. 1992, págs. 148-155. DOI: 10.1109/SPDP.1992.242751.

[13] John L Hennessy y David A Patterson. *Computer Architecture: A Quantitative Approach*. Sixth Edition. Morgan Kaufmann Publishers, 2019. ISBN: 978-0-12-811905-1.

[14] Steven Ho. *RISC-V Instruction Formats*. URL: https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/Lecture7.pdf.

[15] Axel. Jantsch y Hannu. Tenhunen. *Networks on chip*. Kluwer Academic Publishers, 2003. ISBN: 0306487276.

[16] John L. Henessy y David A. Patterson. «Appendix F. Interconnection Networks». En: *Computer architecture: a quantitative approach*. 6.ª ed. 2019. ISBN: 9780128119068.

[17] Maryam Kamali y col. «Towards correct and reusable Network-on-Chip architectures». En: *Modeling and Simulation of Computer Networks and Systems: Methodologies and Applications* (abr. de 2015), págs. 357-392. DOI: 10.1016/B978-0-12-800887-4.00012-2.

[18] Open Source Initiative. *Licenses & Standards*. URL: https://opensource.org/licenses.

[19] RISC-V International. *RISC-V Exchange: Cores & SoCs*. URL: https://riscv.org/exchanges/cores-socs/.

[20] F. Safaei y col. «Pipelined circuit switching: Analysis for the torus with non-uniform traffic». En: *Journal of Systems Architecture* 54.1-2 (ene. de 2008), págs. 97-110. ISSN: 1383-7621. DOI: 10.1016/J.SYSARC.2007.04.004. URL: https://www.sciencedirect.com/science/article/abs/pii/S1383762107000562.

[21] Ultra-Embedded. *biRISC-V - 32-bit dual issue RISC-V CPU*. 2020. URL: https://github.com/ultraembedded/biriscv.

[22] Andrew Waterman, Krste Asanović y RISC-V Foundation. *The RISC-V Instruction Set Manual: Volume I: User-Level ISA, Document Version 20191213*. Inf. téc. Berkeley: CS Division, EECS Department, University of California, dic. de 2019. URL: https://riscv.org/technical/specifications/.

[23] Western Digital. *EH1 RISC-V SweRV CoreTM 1.9*. 2019. URL: https://github.com/chipsalliance/Cores-SweRV.

[24] Western Digital. *EH2 SweRV RISC-V CoreTM 1.4*. 2020. URL: https://github.com/chipsalliance/Cores-SweRV-EH2.

[25] Western Digital. *EL2 SweRV RISC-V CoreTM 1.4*. 2021. URL: https://github.com/chipsalliance/Cores-SweRV-EL2.