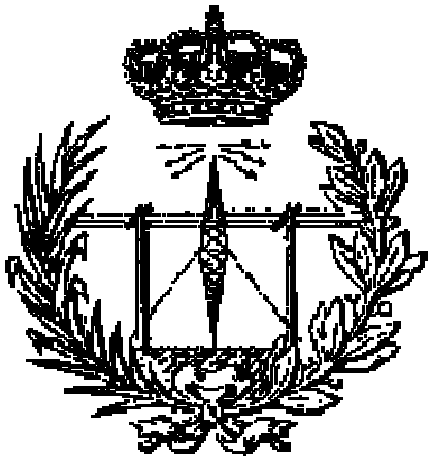
UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN



PROYECTO FIN DE CARRERA

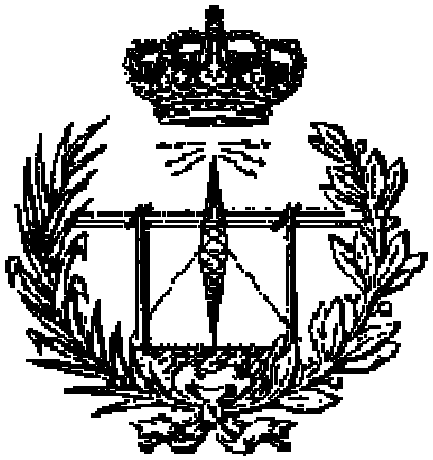
ENTORNO DE MODELADO Y ANÁLISIS RÁPIDO DE SISTEMAS-EN-CHIP MULTIPROCESADOR BASADOS EN NETWORK-ON-CHIP (NOC)

Autora: María Teresa Medina León
Tutores: Dr. D. Félix B. Tobajas Guerrero
D. Víctor Reyes Suárez

Fecha: Diciembre 2006

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN



PROYECTO FIN DE CARRERA

ENTORNO DE MODELADO Y ANÁLISIS RÁPIDO DE SISTEMAS-EN-CHIP MULTIPROCESADOR BASADOS EN NETWORK-ON-CHIP (NOC)

HOJA DE FIRMAS

Alumna

Fdo.: María Teresa Medina León

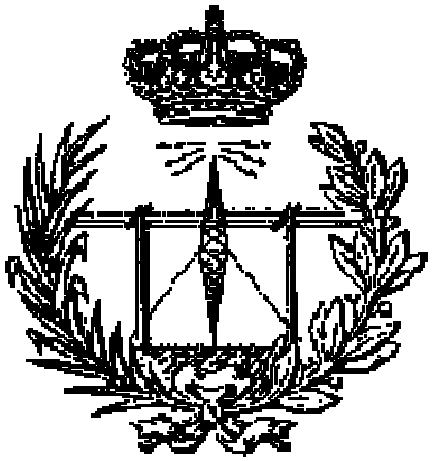
Tutor **Tutor**

Fdo.: Dr. D. Félix B. Tobajas Guerrero Fdo.: D. Víctor Reyes Suárez

Fecha: Diciembre 2006

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN



PROYECTO FIN DE CARRERA

ENTORNO DE MODELADO Y ANÁLISIS RÁPIDO DE SISTEMAS-EN-CHIP MULTIPROCESADOR BASADOS EN NETWORK-ON-CHIP (NOC)

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo.:

Vocal **Secretario**

Fdo.: Fdo.:

Fecha: Diciembre 2006

Agradecimientos

El estar agradecida enormemente a mis tutores: Félix y Víctor, quienes pacientemente me orientaron, y que generosamente me prestaron su tiempo y dedicación durante todo el desarrollo del proyecto. Así como, a sus homónimos holandeses, Bart y Henk con quienes, además, tuve la suerte de poder contrastar ideas y pensamientos. No puedo antes concluir el ámbito académico, sin agradecer profundamente a Valentín, que con su paciencia y predisposición, ha contribuido notablemente en mi formación.

Y para concluir, el eterno reconocimiento a la familia y amigos. A mis padres y hermana, que en todo momento me han mostrado su inquebrantable apoyo, su comprensión, su ánimo, confianza y empuje en los momentos más difíciles. A mi pareja, que inexplicablemente ha soportado mi personal *via crucis*, y a mis amigas, y demás compañeros, con los que he compartido tan buenos momentos.

Peticionario

La redacción del presente Proyecto Fin de Carrera se lleva a cabo a petición de la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universidad de Las Palmas de Gran Canaria, como requisito indispensable para la obtención del título de Ingeniero de Telecomunicación.

Estructura del contenido

Esta primera parte de la memoria se corresponde con el resumen en castellano de la memoria del presente Proyecto Fin de Carrera exigido por resolución vigente de la Comisión de Proyectos Fin de Carrera de la ETSIT, para Proyectos Fin de Carrera realizados en centros extranjeros en el marco de los programas de intercambio reconocidos por la Escuela. La memoria original redactada en inglés se adjunta en la segunda parte de este trabajo. El título del Proyecto desarrollado es *Fast modelling and analysis of NoC-based MPSoCs (Entorno de modelado y análisis rápido de sistemas-en-chip multiprocesador basados en Network-on-Chip (NoC))*, y ha sido realizado en la *Technische Universiteit Eindhoven* (Holanda), en el grupo *Information and Communication Systems (ICS)* del Departamento de *Electrical Engineering*, y dentro del programa de intercambio Sócrates/Erasmus.

La finalidad de esta primera parte no es la de profundizar en el trabajo realizado sino la de detallar brevemente el contenido del trabajo original, así como la exposición de la metodología utilizada y, por último, presentar los resultados obtenidos. Por este motivo y para cualquier duda o necesidad de detalle, se remite al lector a consultar la memoria original en inglés incluida en la segunda parte de este trabajo.

La estructura del presente resumen consta de una introducción al Proyecto Fin de Carrera realizado, en la que además se presentan los objetivos. A continuación se presenta un resumen de cada una de las partes de las que consta este trabajo, incluyendo las conclusiones y posibles líneas de investigación futuras. Por último, se muestra el presupuesto calculado para este Proyecto Fin de Carrera.

PRIMERA PARTE: Introducción, Resumen de la Memoria y Presupuesto

Índice

Índice de Figuras

Capítulo 9

Capítulo 10

Índice de Tablas

Capítulo 1

Introducción

1.1 Planteamiento del problema

Los *Systems-on-Chip* (*SoC*s) son una nueva categoría de sistemas que han emergido durante los últimos años. En dichos sistemas, los procesadores y otros componentes del sistema están disponibles como propiedad intelectual (IPs) y son integrados en un solo chip. Estos IPs (CPUs, DSPs, memorias, periféricos, etc.) incorporan mayor reusabilidad [1-2] y flexibilidad al diseño ya que pueden ser fácilmente personalizados e integrados en múltiples proyectos de diseño.

Actualmente, los *SoC*s están incrementándose en complejidad [3-4] y en los últimos tiempos los diseñadores han comenzado a emplear cada vez más plataformas multiprocesadoras (*MP-SoC*) que combinan varios dispositivos programables. Debido al alto número de IPs que necesitan comunicarse entre sí, no es viable el uso de un simple bus compartido o una jerarquía de buses. Para abordar este problema de interconexión, las tendencias actuales son los *Network-On-Chip* (*NoC*) [5], que consisten en una red compuesta por elementos de conmutación que permiten la conexión de todos los dispositivos de forma que puedan comunicarse entre ellos.

El diseño de *SoCs* no sólo incluye el desarrollo de dispositivos *hardware*, sino también de aplicaciones *softwar*e. Debido al incremento de complejidad y a las limitaciones del mercado, que fuerzan a que el ciclo de desarrollo de IC sea lo más corto

posible, resulta un serio inconveniente esperar a disponer del chip físicamente para comenzar el desarrollo de aplicaciones software. Por tanto, el desarrollo del *software* conviene ser llevado a cabo, o al menos iniciarlo en los primeros pasos del flujo de diseño mediante el uso de simuladores *hardware*.

El modelado *hardware* a nivel de abstracción RTL (*Register Transfer Level*) no permite el comienzo del *software* antes de que el código en lenguaje HDL esté finalizado. Además, la lentitud de las simulaciones utilizando modelos *hardware* descritos a nivel RTL no permite la escabilidad. Esto se debe al nivel de abstracción empleado (RTL), el cual incluye demasiados detalles de la implementación, como protocolos, y otros aspectos irrelevantes para las aplicaciones *software*.

Para abordar estas limitaciones, los diseñadores han elevado el nivel de abstracción de los modelos y han permitido el modelado a nivel de sistema [6]. Así, un nuevo nivel de abstracción denominado *Transaction Level Modeling* (TLM) ha sido introducido en el flujo de diseño, en el cual sólo se modela aquellos aspectos necesarios para que se ejecute el *software* [7]. La rapidez de las simulaciones de estos modelos permite desarrollar con antelación las aplicaciones *software*, además de verificar el hardware del cual depende ésta [8]. La información temporal puede ser incorporada en estos modelos para analizar sus prestaciones y explorar la arquitectura. El resultado son modelos que proporcionan un análisis temprano de las características del sistema antes de acometer el desarrollo a nivel de abstracción RTL.

A nivel TLM, la plataforma está compuesta por un conjunto de módulos conectados entre sí mediante canales de comunicación. A través de estos canales se realizan las transacciones, intercambiándose así datos entre módulos. Estas transacciones de datos pueden ser una simple palabra, una serie de palabras, o complejas estructuras de datos.

TLM es un concepto independiente de un lenguaje. Sin embargo, para implementar y refinar modelos TLM, es útil tener un lenguaje como SystemC [9-13] cuyas características soportan un depurado independiente de la funcionalidad y de las comunicaciones, lo cual es crucial para un desarrollo eficiente a nivel TLM.

SystemC es una librería de C++ dirigida específicamente al modelado a nivel de sistema. Presenta todos los beneficios que posee C++: es un lenguaje orientado a objetos que permite el pleno uso del encapsulado de datos y de conceptos genéricos de programación. SystemC 2.1 facilita el desarrollo de modelos en TLM ya que proporciona módulos, señales para el modelado de comunicaciones a bajo nivel y de sincronizaciones del sistema, además de la noción del tiempo de simulación [14-15].

1.2 Objetivos del Proyecto

El proyecto *Mini-NoC* consiste en una plataforma multiprocesadora (*MP-SoC)* con cuatro nodos procesadores interconectados a través de una red *NoC* [16]. Los componentes de la plataforma están descritos en especificación RTL, por tanto la plataforma es sintetizable. Sin embargo, las simulaciones de la descripción RTL de esta plataforma son extremadamente lentas debido al bajo nivel de abstracción usado en el modelado de los componentes que la forman. Este factor hace que el modelo no sea el más indicado para el análisis de prestaciones y para llevar a cabo actividades de exploración del espacio de diseño, donde múltiples simulaciones se necesitan de forma iterativa. Como consecuencia de esto, y con el objetivo de mejorar la velocidad de simulación y de facilitar el modelado de sistemas multiprocesador, se pretende utilizar el entorno *CASSE*. *CASSE* es un entorno de modelado y simulación a nivel de sistema que proporciona un flujo de diseño que cubre desde la especificación de la aplicación hasta su implementación, a través del modelado de la arquitectura, el mapeado de la funcionalidad en la arquitectura y, el análisis de la funcionalidad y de las prestaciones [18].

El principal objetivo de este Proyecto Fin de Carrera es el de adaptar e integrar los componentes *Mini-NoC* en *CASSE* para permitir el modelado y la evaluación de *MP-SoC* basados en *NoC*. Para ello, se realizará un modelo *CASSE Mixed-Level* en el que los componentes *Mini-NoC* serán adaptados para que usen los protocolos de comunicación a nivel de transacciones disponibles en *CASSE*, que a su vez, también mejorará considerablemente la velocidad de simulación, manteniendo un buen nivel de precisión en los resultados. En el modelo resultante se mapeará una aplicación multimedia con el objetivo de realizar una comparativa entre el modelo *CASSE Mixed-*

Level y la plataforma *Mini-NoC* original. Para ello, es necesario adaptar la aplicación para que use el modelo de programación proporcionado por *CASSE*. Los resultados de la comparativa serán analizados para realizar una evaluación de la velocidad de simulación, precisión, y esfuerzo de modelado, y también para analizar si el entorno *CASSE* permite mejorar la exploración del espacio de diseño.

1.3 Descripción de la Memoria

La estructura del resumen en castellano del presente Proyecto Fin de Carrera corresponde con la seguida en la memoria original en inglés. Está dividida en diez capítulos, uno más que la memoria original en inglés, en el que se introducen conceptos necesarios para la comprensión de las tareas realizadas en este trabajo.

En el presente resumen se presentan, en primer lugar, las actuales limitaciones en el diseño de Sistemas-en-Chip *(SoC)*, y se introduce el concepto de modelado a nivel de sistema a partir de la introducción de un nuevo nivel de abstracción *Transaction Level Modeling* (*TLM*) en el flujo de diseño. Seguidamente, se exponen de forma detallada los objetivos propuestos para este Proyecto Fin de Carrera.

Tras este primer Capítulo de introducción al Proyecto Fin de Carrera y el planteamiento de los objetivos, en el Capítulo 2, "*Estado del arte*" se proporciona una introducción a los Sistemas-en-Chip (*SoC*), *Network on Chip* (*NoC*), y el modelado a nivel TLM, exponiendo los fundamentos de sistemas y redes en chip, y presentando el nuevo nivel de abstracción, necesario para la comprensión de las tareas realizadas en este trabajo. En el Capítulo 3, "*Plataforma Mini-NoC*" se describe la arquitectura y funcionalidad de la plataforma *Mini-NoC*, que es el punto de comienzo de este Proyecto Fin de Carrera. El Capítulo 4, "*Entorno CASSE*" introduce las características de *CASSE*, un entorno de modelado y simulación a nivel de sistema, el cual se ha aplicado para modelar la nueva plataforma Mini-NoC a nivel de TLM. A continuación, en el Capítulo 5, "*Herramienta CTAP*" se describe brevemente la funcionalidad de la herramienta CTAP, empleada para anotar la información temporal en modelos sin información temporal. Finalmente, en el Capítulo 6, "*Aplicaciones*", se introduce la aplicación elegida para realizar la comparativa de prestaciones.

La segunda parte de este resumen comienza con el Capítulo 7, "*Modificaciones en el Mini-NoC"*, en el que se describen las mejoras y novedades introducidas en la plataforma *Mini-NoC*. El Capítulo 8, "Modelo *CASSE Mixed-Level"* presenta los pasos seguidos para desarrollar la nueva plataforma, el modelo *CASSE Mixed-Level*. Por último, el Capítulo 9, "*Comparativa*" presenta los resultados obtenidos de las métricas realizadas sobre cada plataforma.

La última parte de este resumen contiene el Capítulo 10, "*Conclusiones y Futuras Líneas de Trabajo*", donde se introducen las conclusiones y posibles líneas futuras de trabajo. Finalmente, se incluye el presupuesto calculado para la realización de este Proyecto Fin de Carrera.

Capítulo 2

Estado del arte

2.1 Introducción

Hoy en día no podríamos imaginarnos nuestra vida diaria sin los circuitos integrados (IC) puesto que se utilizan extensivamente en productos electrónicos destinados al consumidor, en las telecomunicaciones, la informática, automóviles, productos multimedia, aviones, industrias, etcétera. Esta invención realmente ha realizado grandes cambios en nuestro estilo de vida. Los circuitos integrados son, por esta razón, identificados como una de las invenciones más importantes del siglo pasado.

El diseño de sistemas microelectrónicos se ha estado enfrentando a grandes retos de forma continua debido al constante crecimiento de la complejidad de estos sistemas. Según la Ley de Moore, establecida en el año 1965, la densidad de los chips se doblaría cada 18 meses, sin embargo, desafortunadamente, aún no es posible explotar la cantidad total de transistores disponibles en cada chip. El número de transistores necesarios actualmente para diseñar un IC está claramente por debajo de las posibilidades actuales de un chip.

La *Figura 2.1* muestra la predicción de la diferencia en la productividad del diseño medida como el número de puertas disponibles por chip para una determinada tecnología (línea roja), y el número de puertas usadas por chip (línea azul) en los últimos años.

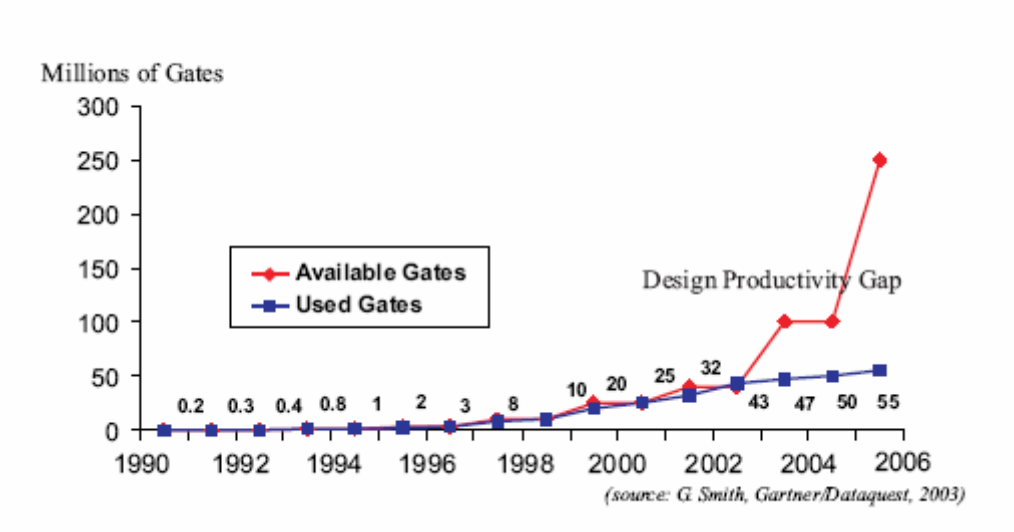


Figura 2.1 : Predicción del vacío en la productividad del diseño [25]

La vida cada vez más corta de los productos, junto con el crecimiento de su funcionalidad, hace que la *tecnología del silicio* pueda alcanzar el billón de transistores en un solo chip a lo largo de esta década [3-4]. Los sistemas integrados aprovecharán esta mejora tan sólo si es posible crear un sistema completo (con elementos procesadores, memorias, infraestructura de comunicación, etc.) en un solo chip (*SoC*).

El diseño de *Sistemas-en-Chip* (*SoC*) se ha convertido en una de las soluciones esenciales para abordar la complejidad de los sistemas y reducir así el *time-to-market*. Sin embargo, el aumento en la heterogeneidad y complejidad de sus actuales y futuras arquitecturas, están forzando a los diseñadores a usar nuevas técnicas. Para abordar estas limitaciones, los diseñadores han aumentado el nivel de abstracción de los modelos y han permitido el modelado a nivel de sistema.

2.2 System-on-Chip

System-on-Chip representa el concepto de concebir e integrar distintos componentes electrónicos en un solo chip para conformar un sistema electrónico completo. El *SoC* se utiliza típicamente en pequeños y complejos productos electrónicos destinados al consumidor como son un teléfono móvil o una cámara fotográfica digital.

Un *SoC* está compuesto de módulos que realizan funciones complejas, los cuales se conocen como *cores* o *Intellectual Property* (IP), y representan el elemento esencial de este tipo de sistemas [1]. Por lo tanto, un *SoC* podría estar compuesto por una gran variedad de *cores* distintos, como pueden ser microprocesadores, grandes bloques de memorias, controladoras gráficas, unidades de procesado digital de señales (DSP), etc. Cada uno de estos *cores* puede estar definido a partir del uso de código sintetizable en un lenguaje de descripción de alto nivel (VHDL, Verilog, SystemC), o a nivel físico desde un *layout* a nivel de transistores. Estos IPs (CPUs, DSPs, memorias, periféricos, etc.) incorporan mayor reusabilidad [2] y flexibilidad al diseño, ya que pueden ser fácilmente personalizados e integrados en múltiples sistemas.

Actualmente, los *SoCs* están incrementándose en complejidad y en los últimos tiempos los diseñadores han comenzado a emplear cada vez más plataformas multiprocesadoras (*Multi-Processor System-on-chip*, *MP-SoC*), que contienen múltiples dispositivos programables (CPUs).

En el diseño de *Sistemas-en-Chip* la interconexión de los núcleos IP se realiza mediante estructuras de comunicación, como buses compartidos, o el empleo de redes de conmutación integradas, denominadas genéricamente *Network-on-Chip* (*NoC*). En el primer caso, como el representado en la *Figura 2.2*, los buses compartidos son totalmente reusables pero presentan el grave inconveniente de permitir únicamente una transacción cada vez, secuencializando así todas las posibles comunicaciones paralelas. Además, el bus no es escalable con el tamaño del sistema, ya que el ancho de banda del mismo es compartido por todos los componentes conectados a él, por lo que la frecuencia de operación del bus decrece a medida que aumenta el número de componentes del sistema debido al incremento de carga que supone en el sistema.

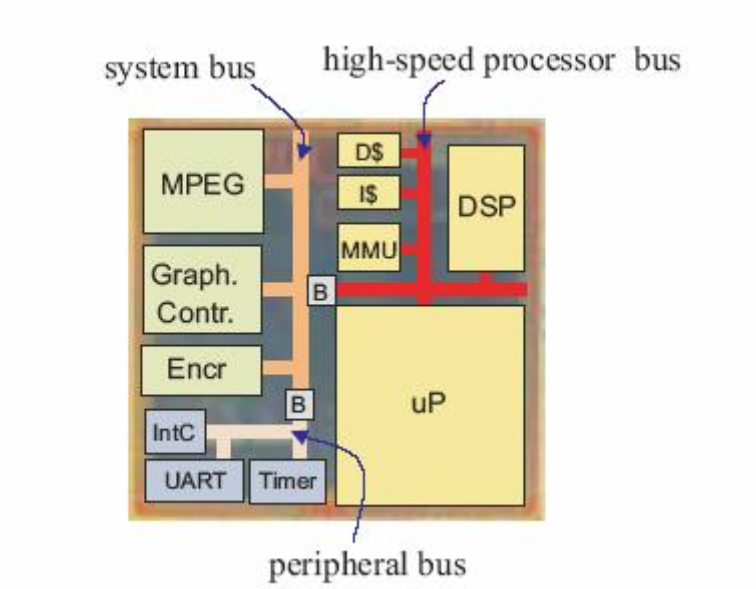


Figura 2.2: Ejemplo de un *SoC* basado en bus

Tratando de encontrar una solución a los problemas presentados por los sistemas basados en buses, algunos investigadores han *tomado prestadas* algunas ideas de las comunicaciones en redes de ordenadores debido principalmente a su capacidad de escalabilidad y su probado éxito. Las redes de conmutación (cuyo principal elemento es el *router*) y las comunicaciones basadas en paquetes como propuesta de comunicación dentro de un chip, representan una buena alternativa a los sistemas basados en buses. La mayoría de los principios y la terminología usada en la comunicación conmutada de paquetes *on-chip* se extraen de teoría de redes de ordenadores. Esta clase de comunicación provee escalabilidad y permite la posibilidad de estandarización y reutilización de esta arquitectura de comunicación. Estas características presentan un menor esfuerzo y permiten a los diseñadores alcanzar las restricciones de *time-to-market*.

2.3 *SoCs* basados en *NoC*

Network on Chip (*NoC*) se está convirtiendo en la solución a las limitaciones que presentan las infraestructuras de comunicaciones basadas en buses. *NoC* se basa en un conjunto de elementos de conmutación o *routers* que componen una red, los cuales permiten una interconexión entre los nodos asociados a los recursos del sistema, o *cores*, de esa red [5][26].

La funcionalidad de un *NoC*, como el que se muestra en la *Figura 2.3*, podría ser la siguiente: un conjunto de datos divididos en paquetes se podrían enviar desde un *IP* origen al canal de entrada de un *router* mediante una conexión (*links*). Esos paquetes son conmutados dentro del *router* hasta el canal de salida correspondiente y conducidos hasta el siguiente *router*. Cuando los paquetes llegan a su elemento de conmutación de destino se encaminan hacia el *IP* correspondiente.

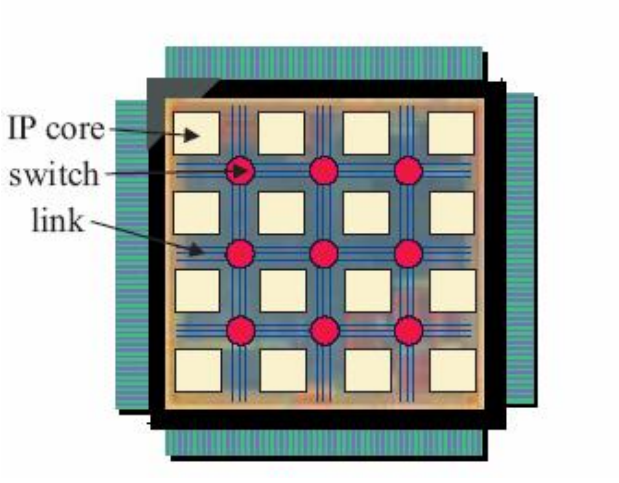


Figura 2.3: Ejemplo de *NoC* con conexiones y conmutadores

Algunas de las características que hacen de los *NoC*s una buena alternativa para solucionar los problemas presentados en los sistemas basados en buses, son las siguientes:

- Transmiten paquetes en lugar de palabras. Dependiendo del algoritmo de encaminamiento, los paquetes pueden ser transferidos a través de la red por distintas rutas, ya que la dirección de destino es parte del paquete. Por lo tanto, no son necesarias líneas de conexión específicas como en el caso de los sistemas basados en buses.

- La infraestructura de comunicación no bloquea el sistema. Haciendo uso de *NoC*, las transacciones de datos se pueden realizar en paralelo si la red está provista de más de un canal de transmisión entre dos nodos de la red.

2.3.1 Redes de interconexión On-Chip

Las comunicaciones *on-chip* se pueden estudiar y clasificar atendiendo a algunos aspectos como pueden ser la topología de interconexión de red, conmutación, ruteado, control de flujo y colas de almacenamiento [27][28]. Se podría entrar en detalle en todos ellos, pero sólo se prestará especial atención a las cuatro primeras características, ya que algunos de estos conceptos se usarán en posteriores capítulos y es interesante tener conocimiento de ellos.

2.3.1.1 Topologías de interconexión de red

Este término define cómo varios nodos están conectados entre sí mediante canales dentro de una red. Normalmente estas topologías se representan como grafos. En la *Figura 2.4* se puede observar una clasificación de las topologías de interconexión de red más populares para arquitecturas multiprocesadores, resaltándose aquellas topologías relacionadas con este Proyecto Fin Carrera.

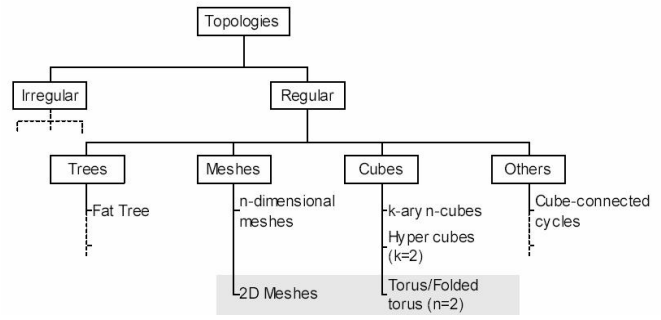


Figura 2.4: Clasificación de las topologías

2.3.1.1.1 Malla n-dimensional

Las *Mallas n-dimensionales* junto a los *k-ary n-cubes* son las topologías de interconexión más utilizadas debido a su regularidad y a su simplicidad de ruteado. En esta topología los nodos se distribuyen en *arrays* n-dimensionales. A lo largo de cada dimensión existen k_i nodos ($k_i \geq 2$), donde $0 \leq i \leq n-1$ es la dimensión. Por lo tanto, el número total de nodos es $k_0 \times k_1 \times \ldots \times k_{n-2} \times k_{n-1}$. Cada nodo x está definido por n

coordenadas (que son las uniones entre los nodos de la *Figura 2.5*), $\sigma_{n-1}(x)$, $\sigma_{n-2}(x)$, …, $\sigma_1(x)$, $\sigma_0(x)$, donde $0 \leq \sigma_i(x) \leq k_i - 1$ para $0 \leq i \leq n-1$. Dos nodos x e y son vecinos si $\sigma_i(x) = \sigma_i(y)$ para cada i, $0 \leq i \leq n-1$, excepto el nodo j, en el que $\sigma_j(x) = \sigma_j(y) \pm 1$. De este modo, los nodos pueden tener de n a $2n$ vecinos dependiendo de su ubicación en la malla, en la que sólo existe comunicación directa entre vecinos.

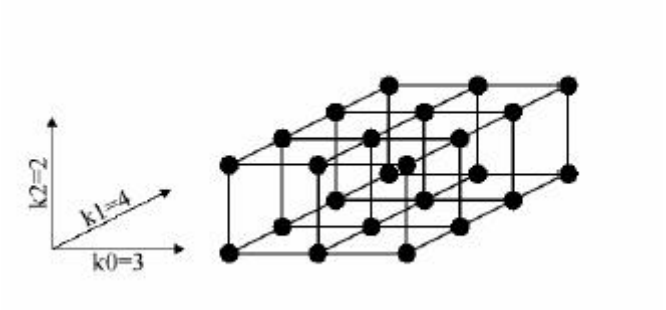


Figura 2.5: Ejemplo de una malla 3-dimensional

El principal problema de las redes de mallas es que, asumiendo un tráfico uniforme entre nodos, los canales del centro de la malla tienden a tener mayor densidad de tráfico que los canales en la periferia.

2.3.1.1.2 k-ary n-cubes

En esta topología los nodos están dispuestos en *arrays* n-dimensionales al igual que en el caso anterior, pero ahora en cada dimensión existe el mismo número de nodos k, por lo que $k_0 = k_1 = \ldots = k_{n-2} = k_{n-1} = k$. El número total de nodos es k_n. Otra diferencia con respecto a las *mallas n-dimensionales* es la definición de nodo vecino. Ahora dos nodos x e y son vecinos si $\sigma_i(x) = \sigma_i(y)$ para cada i, $0 \leq i \leq n-1$, menos uno, j, donde $\sigma_j(x) = (\sigma_j(y) \pm 1) \bmod k$. El uso del módulo en la definición resulta en conexiones entre el último y primer nodo en cada dimensión. Si $k>2$, entonces todos los nodos tienen $2n$ vecinos, mientras que si $k=2$, todos los nodos tienen n vecinos.

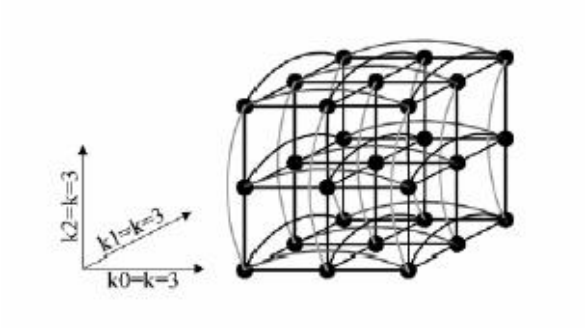


Figura 2.6: Ejemplo de un 3-ario 3-cubo

Debido a su estructura recursiva, se pueden utilizar una gran variedad de grafos (por ejemplo el de mallas) para crear un *k-ary n-cube*, y por esta razón, diversos algoritmos utilizados en esos grafos se pueden aplicar a esta topología. Desafortunadamente, la implementación en dos dimensiones de esta topología proporciona un *layout* ineficiente con una gran densidad de cableado.

2.3.1.1.3 Topología usada

Cuando *n=2*, una malla n-dimensional se reduce a una *malla 2D*, y un *k-ary ncube* se reduce a una topología *torus 2D*. Esta topología de interconexión caracteriza a la plataforma *Mini-NoC* [16] tal y como se expone en el *Capítulo 3*, punto de partida de este Proyecto Fin de Carrera. Esta dimensión es adecuada para implementaciones de redes *on-chip*, ya que cumplen con requerimientos de escalabilidad, *layout* eficiente y eficiencia en energía comparados con otras topologías.

2.3.1.2 Conmutación

Las estrategias de conmutación establecen la forma en la que los paquetes son transferidos desde un nodo origen a un nodo destino, o lo que es lo mismo, cómo atraviesan la red en su camino hacia el nodo de destino. Existen un gran número de estrategias de conmutación que se pueden agrupar en dos grandes grupos: las basadas en conmutación de circuitos, y las basadas en conmutación de paquetes. A continuación se expondrán estos dos casos, siendo el segundo el que se utiliza en este Proyecto Fin de Carrera.

2.3.1.2.1 Conmutación de circuitos

En la conmutación de circuitos se establece un camino entre el origen y el destino antes de comenzar el envío de datos. Para ello, se envía la cabecera junto con información de control, que a medida que atraviesa la red va reservando los enlaces físicos. El camino se libera tras transmitirse el mensaje.

La principal ventaja que ofrece la conmutación de circuitos es que, una vez que el circuito está establecido, se consigue un ancho de banda fijo y garantizado. Sin embargo, por otro lado presenta numerosas desventajas, entre las que destacan los elevados tiempos de establecimiento de la conexión, la necesidad de un reloj global y poca flexibilidad.

2.3.1.2.2 Conmutación de paquetes

En la conmutación de paquetes toda la información es transferida a través de la red desde el nodo origen al nodo destino en forma de paquetes, que suelen ser de tamaño variable. Además de la información que el usuario desee transferir de un nodo a otro, cada paquete encapsula información adicional usada para su encaminamiento individual a través de la red, control de flujo, detección de errores, etc. Los paquetes se transmiten de forma asíncrona por la red, lo que quiere decir que cada nodo decide cuándo se ha de transmitir el paquete hacia el nodo siguiente. Cuando el paquete está listo para ser enviado, la fuente trata de obtener ancho de banda de la red, y cuando lo obtiene se transfiere el paquete. Además, los paquetes pueden seguir una misma ruta o no, en cuyo caso es preciso reordenarlos.

Esta estrategia es más flexible que la anterior en lo referente a la asignación del ancho de banda, ya que puede soportar requerimientos de ancho de banda que cambien dinámicamente. El ancho de banda se asigna únicamente cuando el paquete está listo para ser transmitido, y permanece asignado sólo durante el tiempo necesario para transferir el paquete. Sin embargo, la desventaja más crítica en conmutación de paquetes es la ausencia de garantías de ancho de banda, lo que conlleva retardos variables en la comunicación.

2.3.1.2.3 Wormhole routing

Es una las estrategias de conmutación de paquetes más utilizadas en las arquitecturas *NoC*. Esta estrategia se basa en dividir el mensaje que se desea transferir de un nodo a otro en partes más pequeñas llamadas *flits* (unidad básica de control de flujo). Así sólo es necesario almacenar uno o un número reducido de *flits* en cada *router* a la espera de ser transferidos al siguiente nodo. Por lo tanto, no se precisa esperar hasta que haya llegado todo el mensaje para poder comenzar a reenviarlo hacia el siguiente *router*. El primer *flit* enviado por la red establece, a medida que es transferido a través del *NoC*, un camino que seguirá el resto, Dicho camino será cerrado por el último *flit* del paquete (cola).

El principal inconveniente de este método es que, desde que el primer *flit* establece un camino, todos esos recursos de la red permanecerán en uso hasta que el último *flit* los libere. Por lo tanto, si un *flit* bloquea uno de esos canales físicos, éste estará inactivo y no podrá ser usado por otros.

Por otro lado, este método tiene la ventaja de que sólo es necesario enviar información del nodo destino en el primer *flit*, ya que el resto de ellos tan sólo seguirá el camino anteriormente establecido por el *flit* de cabecera. Es por este motivo que *Wormhole* permite transferir únicamente datos en los *flits* que siguen al primero, permitiendo un mayor aprovechamiento del ancho de banda.

2.3.1.3 Encaminamiento

El ruteado, o encaminamiento, implica seleccionar una ruta entre un nodo origen y un nodo destino en una topología determinada. Según sus características, las estrategias de encaminamiento pueden clasificarse de dos formas: dependiendo de dónde se tome la decisión de encaminamiento o dependiendo de en qué se basa la decisión de encaminamiento [29]. A continuación se detallan las características de cada uno de los grupos de estrategias de encaminamiento.
Dependiendo de dónde se tome la decisión de encaminamiento:

- **Encaminamiento distribuido**

  Cada *router* decide por dónde debe ser transferido el paquete que acaba de recibir (si dejarlo en el procesador local o entregarlo a un *router* vecino). En este tipo de encaminamiento cada nodo debe tener una tabla o un algoritmo de encaminamiento que determine la ruta de cada paquete en función de un nodo origen y un nodo destino.

  Presenta la ventaja de poder proporcionar cierta adaptabilidad en función del estado de la red. Así, basándose en información del estado de la red, selecciona de entre todos los puertos de salida el más apropiado. Sin embargo, estas ventajas se logran a costa de una pérdida en el rendimiento, ya que en cada nodo de la ruta se debe acceder a la tabla de encaminamiento para seleccionar la salida apropiada, aumentando así la latencia.

- **Encaminamiento desde el origen**

  El nodo de origen selecciona la ruta por la que deben transferirse los paquetes antes de enviar el primero de ellos. Cada paquete ha de llevar esta información de ruta, por lo que aumenta el tamaño del paquete.

  La ventaja más importante que ofrece este tipo de encaminamiento es la velocidad, ya que tras seleccionar la ruta, no se invierte más tiempo en el cómputo del enrutamiento. Sin embargo, la ruta no se puede cambiar una vez el paquete haya entrado en la red, por lo que presentan el inconveniente de no poder adaptarse a las situaciones de congestión que puedan irse produciendo. Asimismo, a medida que aumenta el número de nodos en la red y los puertos de salida en el elemento de conmutación, se incrementa la capacidad de almacenamiento requerida.

  Dependiendo de en qué se basa la decisión de encaminamiento:

  - **Encaminamiento determinista**

  La ruta queda completamente fijada (y es siempre la misma) tan sólo con los nodos de origen y destino de cada paquete. El encaminamiento determinista es

relativamente simple pero no puede reaccionar ante la congestión de la red o la pérdida de un nodo. Un ejemplo es el algoritmo de encaminamiento *E-cube*, en el que los datos siempre se encaminan sobre las dimensiones de la red en un orden fijo: en primer lugar en la dirección X y luego en la dirección Y.

- **Encaminamiento adaptativo:**

La ruta para un paquete determinado queda fijada por el nodo origen, el nodo de destino y por el estado actual de la red (como por ejemplo posibles congestiones). Para tomar esta decisión, los *routers* deben tener información de la congestión global de la red. Para ello, se añade más tráfico a la red y cada *router* debe tener capacidad de almacenar esta información adicional.

El encaminamiento utilizado por la plataforma *Mini-NoC* de este Proyecto Fin de Carrera es distribuido y determinista, siendo el algoritmo utilizado *E-cube* sobre una topología *2D-Torus.*

2.4 Metodologías de diseño

Actualmente la mayoría de los *SoC*s están compuestos por más de un procesador, por lo que el software se ha convertido en una parte esencial de su diseño. Tradicionalmente, el diseño de los sistemas digitales electrónicos embebidos siguen los pasos del flujo de diseño de la *Figura 2.7*, donde el software del sistema embebido se desarrolla independientemente del diseño hardware.

En el flujo de diseño clásico el trabajo hardware comienza en el desarrollo del código RTL a través de la creación de modelos hardware empleando lenguajes de descripción hardware como VHDL o Verilog. Estos modelos se verifican para asegurar su correcta funcionalidad, para a continuación, realizar la síntesis lógica y obtener la netlist, y obtener finalmente un prototipo del sistema. Una vez que el prototipo está disponible, comienza el desarrollo del software, que una vez terminado será integrado en el sistema y validado.

Sin embargo, este flujo se diseño presenta ciertos inconvenientes. Por un lado, el incremento de complejidad y las limitaciones del mercado fuerzan a reducir el ciclo de desarrollo de un chip, por lo que esperar a que éste exista físicamente para comenzar el desarrollo de su software representa un serio inconveniente. Por otro lado, los diseñadores de sistemas necesitan explorar, en las primeras etapas del flujo del diseño, diferentes protocolos de comunicación y configuraciones, con el fin de tomar las decisiones correctas y eliminar posibles cuellos de botella en el sistema. En la actualidad los *SoCs* son muy complejos [3-4], por lo que no sólo se invierte mucho tiempo en su desarrollo a nivel RTL, sino que las simulaciones resultan extremadamente lentas para permitir la realización de exploraciones relevantes del funcionamiento. En este sentido, la velocidad de las simulaciones afectan no solamente a la exploración adecuada del espacio de diseño, sino que pequeños cambios en el diseño requieren de un considerable esfuerzo de simulación debido a la complejidad natural de estos sistemas.

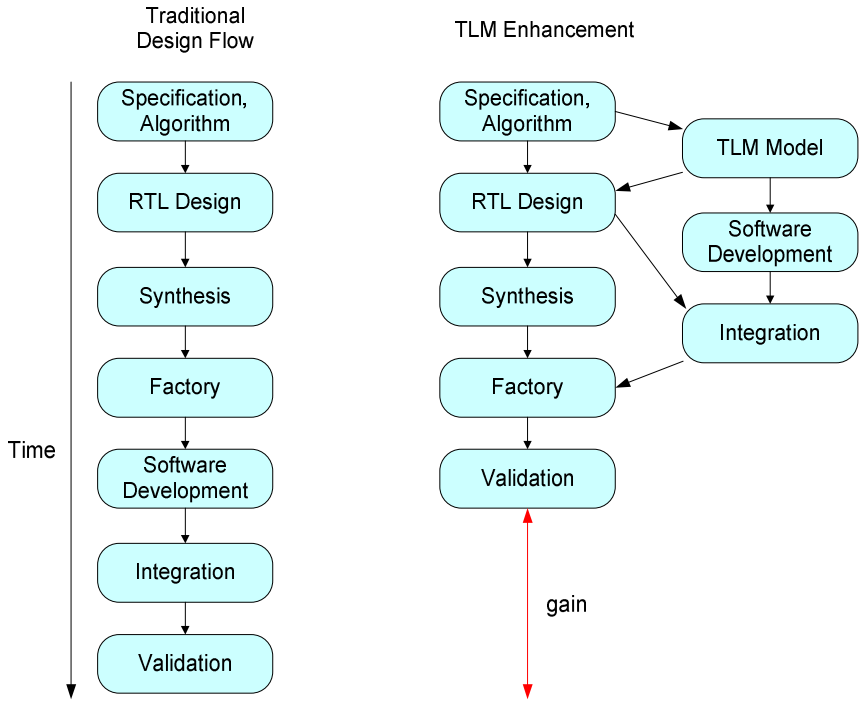


Figura 2.7: Flujo de diseño de *SoC*

Para solventar estas dos limitaciones, los diseñadores han elevado el nivel de abstracción de los modelos introduciendo un nuevo nivel en el flujo denominado *Transaction Level Modeling* (TLM) [7]. La principal diferencia con respecto al flujo

clásico es que ahora se permite el co-diseño del *hardware* y el *software* (ver *Figura 2.7*). Esto es posible debido a que los modelos TLM se modelan rápidamente, por lo que tan pronto como la plataforma TLM está completada, sirve de modelo de referencia único para el desarrollo del software y la plataforma RTL. Esto significa que el desarrollo del software se realiza en paralelo con el desarrollo RTL, y no después como el flujo de diseño clásico, permitiendo así reducir el ciclo de desarrollo del chip.

La rapidez en las simulaciones es alcanzada reduciendo al mínimo el número de eventos y la cantidad de información que tiene que ser procesada durante la simulación. En el nivel de abstracción TLM, la arquitectura de los IPs son modelados a nivel funcional y los buses del sistema se modelan como canales, independientemente de la arquitectura del bus o del protocolo. La reducción de los detalles del diseño y su encapsulado a nivel de transacciones, permite el diseñador modelar de forma sencilla, consiguiendo una percepción rápida de la funcionalidad del diseño y una estimación de sus características. Por tanto, los modelos TLM, gracias a su rapidez en las simulaciones y a la sencillez de sus modelos, permiten una exploración rápida del espacio de diseño antes de acometer su decisión hardware a nivel RTL.

2.5 Distintos niveles de abstracción

En los modelos TLM los detalles de la comunicación entre componentes computacionales están separados de los detalles de dichos componentes. Los detalles innecesarios de ambos, comunicación y cómputo, se omiten en un modelo TLM, siendo posible agregarse con posteridad.

Para simplificar el proceso de diseño, los diseñadores generalmente utilizan un número de modelos intermedios. Estos modelos intermedios dividen el diseño en varias etapas más pequeñas, en las que los modelos pueden ser simulados, estimados y validados independientemente.

Para relacionar los diversos modelos, introducimos el *system modeling graph* (ver *Figura 2.8*) [30]. El eje X de la gráfica representa la computación y el eje Y

representa la comunicación. Cada eje tiene tres grados de precisión: *untimed, approximate-timed,* y *cycle-timed*. Donde:

- *Un-timed computation/communication:* representa la funcionalidad pura del diseño sin ningún detalle de implementación.

- *Approximate-timed computation/communication:* contiene detalles de la implementacion a nivel de sistema, como la arquitectura, mapeado, y los elementos de procesamiento elegidos. El tiempo de simulación es estimado.

- *Cycle-timed computation/communication:* contiene todos los detalles de implementación.

En [7] se definen seis modelos de la abstracción en el modelado a nivel de sistema, indicados con círculos en la *Figura 2.8*. Entre ellos, se encuentran los modelos TLM, indicados por los círculos B, C, D, E: *component-assembly*, *bus-arbitration*, *bus-functional*, y *cycle-accurate*, respectivamente

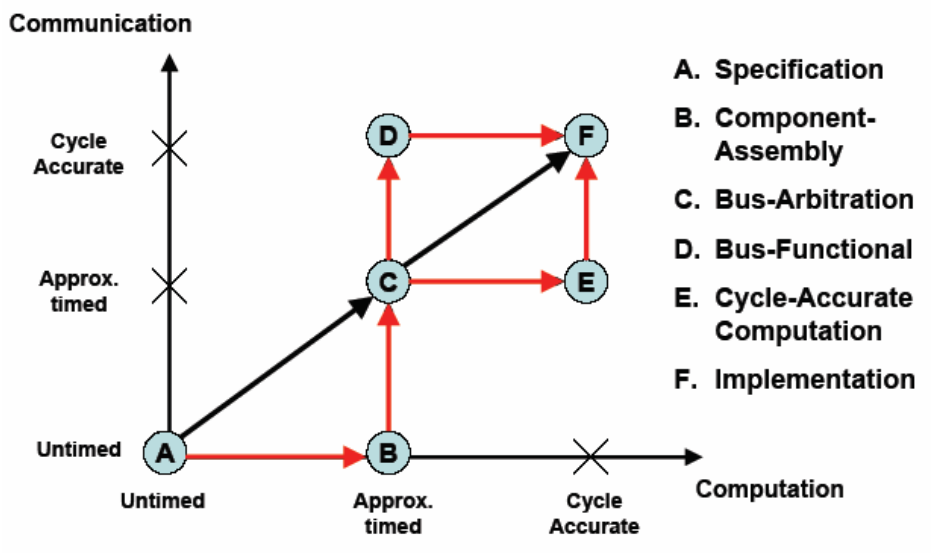


Figura 2.8: ***System modeling graph***

A. Modelo de especificación: describe la funcionalidad del sistema y está libre de cualquier detalle de implementación.

B. **Modelo *component-assembly***: los canales se limitan simplemente a pasar los mensajes, es decir, sólo se representan los datos a transferir o los procesos de sincronización entre elementos de procesamiento (PE), sin ninguna implementación de bus o protocolo; por tanto, la parte del modelo computacional (canal) es *untimed*. Por otro lado, la parte del modelo computacional es *timed*, siendo calculada a partir de la aproximación del tiempo de ejecución de un elemento de procesamiento específico. El tiempo estimado en la computación es calculado a través de un estimador (ver *Capítulo 4*) y la información temporal es anotada en el código insertando declaraciones de tipo *wait().*

C. **Modelo *bus-arbitration:*** la comunicación y la computación del modelo están estimadas temporalmente (*approximate time).* Los canales han sido anotados temporalmente a partir de una declaración *wait()* por transacción.

D. **Modelo *bus-functional*:** la comunicación tiene precisión temporal exacta (*cycle-accurate)* y la precisión computacional estimada (*approximate-timed).* En este modelo los canales son reemplazados por canales con protocolos de precisión *cycle-accurate* representados por señales donde los datos son transferidos por protocolo.

E. **Modelo *cycle-accurate computation*:** la computación tiene precisión *cycle-accurate* y la comunicación *aproximate-timed*. Los elementos de procesamiento están modelados en términos de *cycle-accurate* a través de simuladores de juego de instrucciones (ISS). Este modelo puede ser generado a partir de un modelo *bus-arbitration.*

F. **Modelo de implementación:** tanto la comunicación como la computación tienen precisión *cycle-accurate*. Los componentes están definidos a partir de simuladores de juego de instrucciones (ISS). Este modelo puede ser generado a partir del modelo *cycle-accurate o bus-functional.*

2.6 Conceptos y terminología

Un sistema descrito a nivel TLM consiste en un conjunto de componentes modelados a través de una jerarquía de **módulos** que contienen procesos, puertos, y canales. El comportamiento de los distintos módulos es modelado a partir de **procesos concurrentes o *threads***, que pueden ejecutarse en paralelo.

La comunicación entre módulos se realiza a través de **canales**, tal y como se representa en la *Figura 2.9*. Dependiendo del nivel de precisión requerido, un canal puede ser un simple *router*, un bus, un *network-on-chip*, u otras estructuras. Un canal implementa una o más **interfaces**, donde una interfaz se entiende como un conjunto de funciones de acceso al canal.

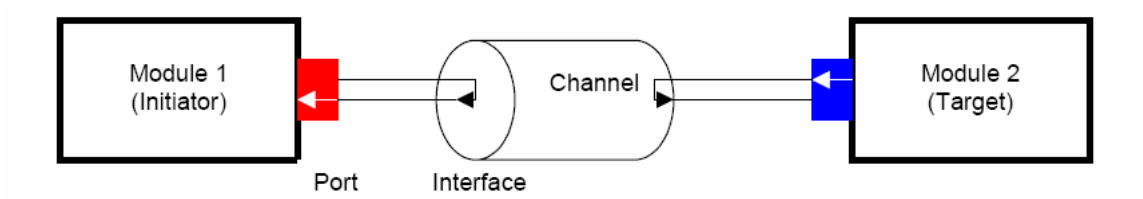


Figura 2.9: Módulos *Initiator* y *Target* conectados a través de un canal de comunicación.

Los módulos y los canales se unen mediante **puertos** de comunicación. Dichos puertos están ligados a un tipo de interfaz, por lo que para conectar un canal a un puerto, el canal debe implementar la interfaz a la que el puerto está asociado.

Las transacciones se realizan llamando a las funciones de las interfaces de los canales modelados, entendiendo una transacción como el conjunto de datos que está siendo intercambiado. Un módulo ***master*** o ***initiator*** es el que inicia las transacciones del sistema, mientras que un módulo ***slave*** o ***target*** es el que recibe y gestiona las peticiones de transacción.

2.7 SystemC

TLM es un concepto independiente de un lenguaje. Sin embargo, para implementar y refinar modelos TLM, es útil tener un lenguaje como SystemC [9-13].

SystemC se ha convertido en un estándar en el diseño a nivel de sistema. La capacidad de SystemC de alcanzar niveles de abstracción más elevados que los leguajes de descripción hardware (HDLs), siendo capaz al mismo tiempo de representar estructuras *hardware*, hacen de él un lenguaje atractivo para la industria.

SystemC 2.1 facilita el desarrollo de modelos a nivel TLM ya que proporciona un amplio conjunto de primitivas de comunicación y sincronización – canales, puertos, eventos, señales, etcétera. La ejecución concurrente se realiza a través de múltiples hilos (*threads*) y procesos, y siendo la ejecución controlada a través de un planificador. Puesto que es una librería de C++, está orientado a objetos, es modular y permite la encapsulación de datos, características todas ellas esenciales para facilitar la distribución del IP, su reutilización y adaptabilidad a través de distintos niveles de abstracción [14-15].

Capítulo 3

La Plataforma *Mini-NoC*

3.1 Introducción

Este capítulo describe la arquitectura y funcionalidad de la plataforma *Mini-NoC*. Comienza con un breve repaso a la arquitectura de la plataforma, detallando a continuación la funcionalidad de cada módulo, y describiendo cómo se lleva a cabo la comunicación entre nodos. El capítulo termina con una sección en la que se explica la forma en la que se ejecuta una aplicación en la plataforma *Mini-NoC*.

3.2 Arquitectura

La plataforma *Mini-NoC* está compuesta por cuatro procesadores *mMIPS* [16] dispuestos en cuatro nodos comunicados entre sí mediante una red *mNoC* provista de *routers*. Este *Network-On-Chip* presenta una topología *torus* e implementa un encaminamiento determinista basado en el algoritmo *E-cube*. Inicialmente, la implementación de la plataforma *Mini-NoC* está realizada en C++, haciendo uso de librerías SystemC [14], y todas las aplicaciones usadas para probar la funcionalidad del sistema se han descrito en lenguaje C.

Los cuatro procesadores *mMIPS* acceden a la red *mNoC* a través de un módulo bidireccional denominado *Network Interface* (*NI*). Existe un módulo intermedio entre el procesador y la interfaz de red, denominado *MEMDEV*, que controla los accesos a la memoria de datos del *mMIPS* y realiza las comunicaciones con la interfaz de red.

La comunicación entre dos nodos a través de la red se realiza mediante paso de mensajes a partir del uso de dos funciones *software*: *sc_send()* y *sc_receive()*. Estas funciones están implementadas en la librería *stdcomm*.

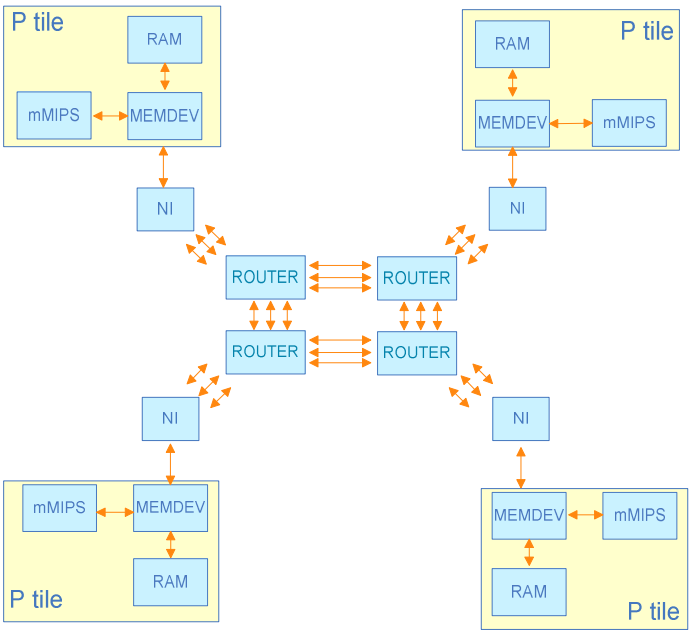


Figura 3.1: Plataforma *Mini-NoC*

3.2.1 *mMIPS*

El procesador *mMIPS* (mini MIPS) es una versión simplificada de un procesador MIPS [17]. Es un sistema segmentado realizado en código SystemC sintetizable. Comparado con el procesador MIPS tradicional, *mMIPS* presenta un juego reducido de instrucciones de 28 instrucciones, con lo que algunas operaciones deben ser realizadas por software (operaciones de punto flotante, multiplicación, división, modulo, etcétera) y, por tanto, se requiere de un mayor tiempo de ejecución.

3.2.2 *MEMDEV* y *Network Interface*

Los procesadores *mMIPS* están comunicados entre sí a través de la red en-chip mediante sus Interfaces de Red (*Network Interface*, *NI*). Se puede acceder a estos módulos a través de ciertas direcciones de memoria, estando el *NI* controlado por el *mMIPS* a través de directivas *load/store* sobre dichas posiciones de memoria. Otro

módulo, denominado *MEMDEV*, es el que interpreta y detecta cuándo los accesos a memoria son a la RAM local o se intenta establecer una comunicación a través del *NI*, dependiendo de a qué direcciones de memoria se intenta acceder. Esta configuración se refleja en la *Figura 3.2*.

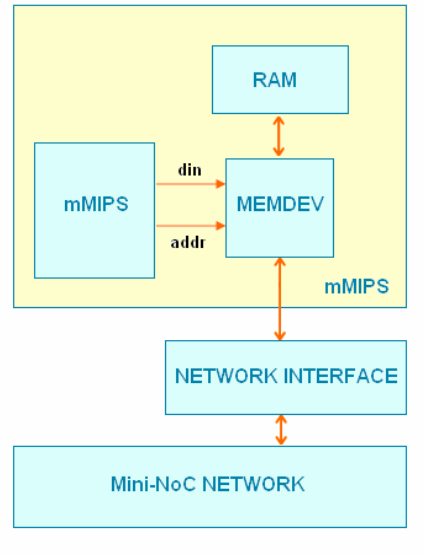


Figura 3.2: El módulo *MEMDEV* accede a la memoria RAM o al *NI* dependiendo del valor de la dirección

El módulo *MEMDEV* es capaz de reconocer dos direcciones asignadas al *NI*: 0x80000000 y 0x80000004. La primera dirección (dirección de palabra de datos, *DW*) está asociada a los datos del *NI*, mientras que la segunda (dirección de palabra de control, *CW*) se utiliza para el control del *NI*. Leyendo o escribiendo de la dirección *DW* se produce una lectura o escritura en los buffers internos del *NI*. Para controlar el estado del *NI*, se puede acceder a la dirección *CW* a través de la dirección 0x80000004. Es importante mencionar el significado de cada uno de los bits que componen la palabra de control, representadas en la *Figura 3.3*.

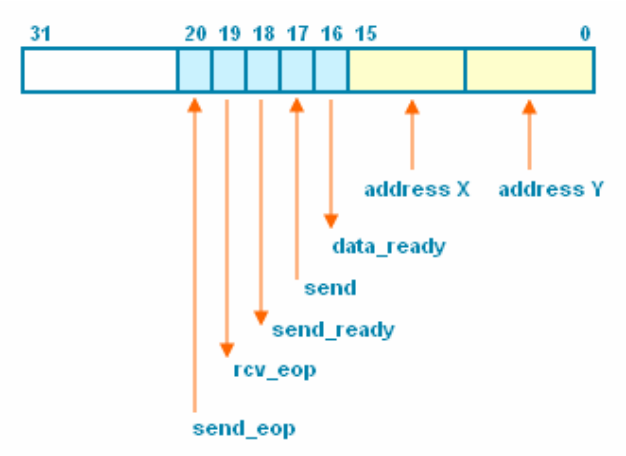


Figura 3.3: Significado de los bits de la *CW* del *NI* (0x80000004)

Atendiendo la *Figura 3.3*, se puede apreciar que algunos bits de la palabra de control son sólo de escritura (0-15, 17 y 20 – controlan el comportamiento del *NI*) mientras que otros son sólo de lectura (16, 18 y 19 – informan del estado del *NI*). Cabe reseñar también que los bits 21 al 31 están libres y podrían ser usados para otros propósitos.

Los bits de estado del *NI* son los siguientes:

- *data ready* (bit 16) – informa de que ha llegado un dato nuevo al *NI* y que está listo para ser leído. Este bit volverá a estar inactivo cuando el dato haya sido leído.
- *send ready* (bit 18) – el dato anterior ya ha sido enviado y el *NI* preparado para enviar otro dato.
- *received end of packet* (bit 19) – el *NI* puede leer paquetes multipalabra al recibir de forma consecutiva palabras de 32 bits. Si este bit está activo, junto con el bit *data ready*, indica que la palabra leída es la última del paquete.

Los bits de control del *NI* se presentan a continuación:

- *address bits (bits 0-15)* – Usados para escribir la dirección relativa de destino del paquete (distancia X - bits [15:8], distancia Y - bits [7:0]).
- *send bit* (bit 17) – Activando este bit se activa el envío del dato escrito en la *DW* hacia la dirección establecida mediante los bits *address bits*.
- *send end of packet* (bit 20) – Activando este bit, junto con el bit *send bit*, se indica que la palabra escrita en la *DW* es la última del paquete.

3.2.3 La red *Mini-NoC*

La red que conecta los procesadores *mMIPS* en el *mNoC* es una red *torus* basada en un algoritmo de encaminamiento *E-cube*. La configuración usada en el *mNoC* original es de dos nodos de alto por dos nodos de ancho.

En una topología *torus*, cada nodo de la red está conectado a su vecino más próximo en ambas dimensiones. En los bordes de la red, el último nodo está conectado al primero en una dimensión dada, como se representa en la *Figura 3.4*.

En el algoritmo de encaminamiento *E-cube*, cada paquete enviado a través de la red es encaminado primero a lo largo de la dimensión X hasta que llega al *router* localizado en la vertical del nodo destino. A continuación, comienza a desplazarse a lo largo de la dimensión Y hasta que alcanza el *router* asociado al nodo destino.

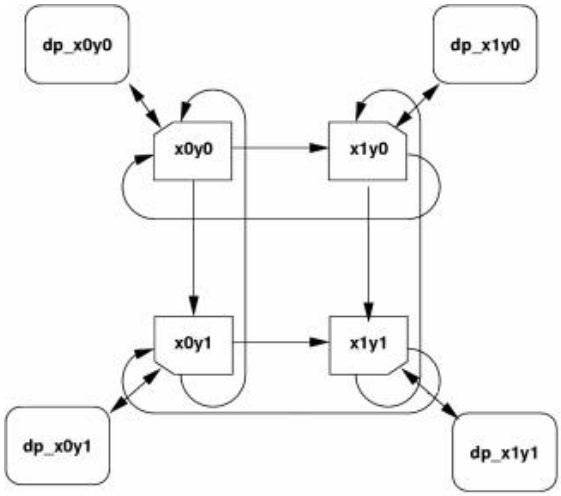


Figura 3.4: Cuatro routers *(xXyY)*, cuatro *mMIPS* (*dp_xXyY*) y las líneas de datos que los conectan

3.3 Librerías para el *Mini-NoC*

Existen dos librerías disponibles para la creación y el depurado de código C en la plataforma *Mini-NoC*: *stdcomm* y *mtools* (ver detalle en sección 2.3.2 de la Memoria).

3.3.1 Librería C para el paso de mensajes (*stdcomm*)

La librería *stdcomm* contiene la implementación del protocolo de comunicación por paso de mensajes para el *Mini-NoC*. Las siguientes dos secciones presentan las primitivas usadas en el código C de las aplicaciones ejecutadas por los *mMIP*S, las cuales son interpretadas por el compilador a fin de proporcionar las instrucciones apropiadas para el acceso al *NI* por parte del *mMIPS*.

3.3.1.1 Primitiva *sc_send()*

La primitiva *sc_send()* se usa para enviar datos. Consta de tres parámetros: la dirección relativa del nodo destino, un puntero hacia el *buffer* donde están almacenados los datos a enviar, y el tamaño en bytes de estos datos:

```
int sc_send(const int address, const void *data, const int size_in_bytes);
```

La primitiva *sc_send()* divide el mensaje en palabras de 32 bits y utiliza otra primitiva, denominada *sc_send_word()* para enviar cada una de las palabras. La primitiva *sc_send_word()* chequea si la interfaz de red está lista para el envío antes de enviar cada palabra.

3.3.1.2 Primitive *sc_receive()*

La primitiva *sc_receive()* se emplea para recibir datos. En este caso sólo son necesarios dos parámetros: un puntero hacia el buffer donde el dato recibido debe almacenarse, y el tamaño en bytes del dato a recibir.

```
int sc_receive( const void *data, const int size_in_bytes);
```

La primitiva *sc_receive()* se encarga de recibir todo el mensaje. Reconstruye el mensaje mediante el uso de otra primitiva, denominada *sc_receive_word()*, que se encarga de recibir cada palabra de 32 bits. La primitiva *sc_receive_word()* comprueba si el *NI* tiene la palabra de datos lista antes de ser leída.

3.4 Ejecutando una aplicación en la plataforma

Esta sección explica cómo se ejecuta una aplicación en la plataforma *Mini-NoC* y además, presenta las distintas formas de obtener información temporal una vez que la aplicación se ha ejecutado.

Cuando se desea ejecutar una aplicación en la plataforma *Mini-NoC*, el primer paso es compilar dicha aplicación y la librería *stdcomm* con el compilador *LCC*. El siguiente paso, es ejecutar ambos en el simulador de SystemC para el *Mini-NoC*. El *Mini-NoC* está listo para usarse simplemente compilando todos sus ficheros fuente con el compilador *GCC* C++, como se muestra en la *Figura 3.5*.

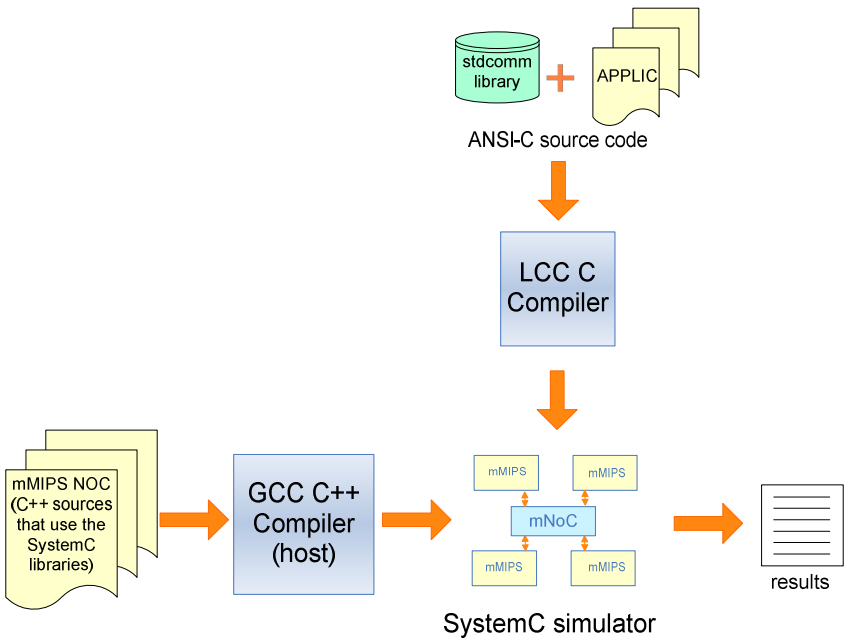


Figura 3.5: Proceso seguido para ejecutar una aplicación en la plataforma *Mini-NoC*

Cuando la ejecución termina, se puede obtener información temporal adicional mediante un fichero que recoge la información mostrada en la salida estándar durante la ejecución. A partir de la información obtenida de este fichero se pueden observar los pasos seguidos por los datos a través de todos los módulos que implementan el *mNoC*, desde que son enviados desde un procesador *mMIPS*, hasta que alcanza el nodo destino.

Las fuentes del *Mini-NoC* proporcionan el tiempo total simulado en el sistema a partir del fichero *main_net.cpp*, del cual se presenta un extracto en la *Figura 3.6*. Dicho fichero incorpora la función *sc_time_stamp()* de SystemC (ver línea 16), cuya salida puede ser leída en la salida estándar cuando la ejecución de la aplicación termina. También se incluye dicha función en cada nodo del sistema, permitiendo conocer el instante en el que cada nodo de la plataforma ha finalizado su ejecución (ver línea 13). El reloj del sistema se implementa también en dicho fichero (ver líneas 5 y 10) con un período de 10 nanosegundos.

```
1  //unnecessary lines omitted
2  #include <time.h>
3  …
4  sc_signal<bool> clk;
5  unsigned sim_time = 0, period = 10;
6  int sc_main(int argc, char *argv[]){
7  …
8  while( max_time < 0 || sim_time < (unsigned)max_time ){
9    clk = 0;
10   sc_cycle(period/2);
11   …
12  if( !e01 && dp_x0y1_pc.read().to_uint() == 0x28 ) {
13     cout << "FINISHED x0y1 @ " << sc_time_stamp() << endl;
14     e01 = true; }
15  if( e00 && e10 && e01 && e11 ) {
16     cout << "ALL FINISHED @ " << sc_time_stamp() << endl;
17     break;
18  …
```

Figura 3.6: Extracto del fichero *main_net.cpp*

Al finalizar la simulación se genera el fichero *mips.vcd*, que contiene la traza de todas las señales del sistema a lo largo de la simulación. Con este fichero es posible acceder a los retardos temporales asociados a todos los módulos que conforman la plataforma *Mini-NoC*. Este fichero puede ser visualizado a través del visualizador de formas de onda GTKWave, cuya herramienta genera las formas de onda permitiendo examinar los resultados almacenados en los ficheros *.*vcd* después de la simulación.

Capítulo 4

Entorno *CASSE*

4.1 Introducción

CASSE es un entorno de modelado y de simulación a nivel de sistema que ha sido desarrollado en la Universidad de Las Palmas de Gran Canaria. [18]. Proporciona un flujo de diseño que cubre, desde la especificación de la aplicación, hasta su implementación, a través del modelado de la arquitectura, el mapeado de la funcionalidad en la arquitectura, y el análisis de la funcionalidad y las prestaciones.

La herramienta tiene como objetivo facilitar y acelerar el proceso de modelado y análisis de los *MP-SoCs.* Esto es posible gracias a que el entorno *CASSE* permite comenzar a modelar con elementos predefinidos que pueden crearse y configurarse mediante ficheros de especificación.

Este capítulo presenta una breve introducción al entorno *CASSE*, donde únicamente se describen sus características generales y las diferentes etapas seguidas en la herramienta para modelar un sistema. Sin embargo, si se desea, se puede consultar más información acerca de la herramienta en [18-21]. Este Proyecto Fin de Carrera empleará el entorno *CASSE* para modelar un nuevo sistema, que se describe en detalle en el *Capítulo 7* del presente documento.

4.2 Metodología de diseño

El entorno *CASSE* sigue una típica metodología *Y-chart* [6] en la que la funcionalidad de la aplicación y la arquitectura son modeladas independientemente y combinadas en una fase separada de mapeado.

Una de las características del entorno *CASSE* es que el usuario guía todas las etapas del flujo de diseño (modelado de la aplicación, modelado de la arquitectura, mapeado y simulación) mediante ficheros de texto. Estos ficheros descriptivos son leídos e interpretados por la herramienta durante el tiempo de elaboración con el fin de crear y configurar apropiadamente el modelo de sistema deseado. El resultado es un modelo ejecutable que se ejecuta usando el *kernel* de SystemC. Por tanto, la herramienta simplifica la exploración del espacio de diseño mediante estos ficheros de texto descriptivos, que pueden ser modificados fácilmente para crear un nuevo sistema. Debido a que cambiar los ficheros de descripción no requiere recompilar los modelos existentes, se puede realizar un barrido extenso de parámetros de forma fácil usando *scripts*.

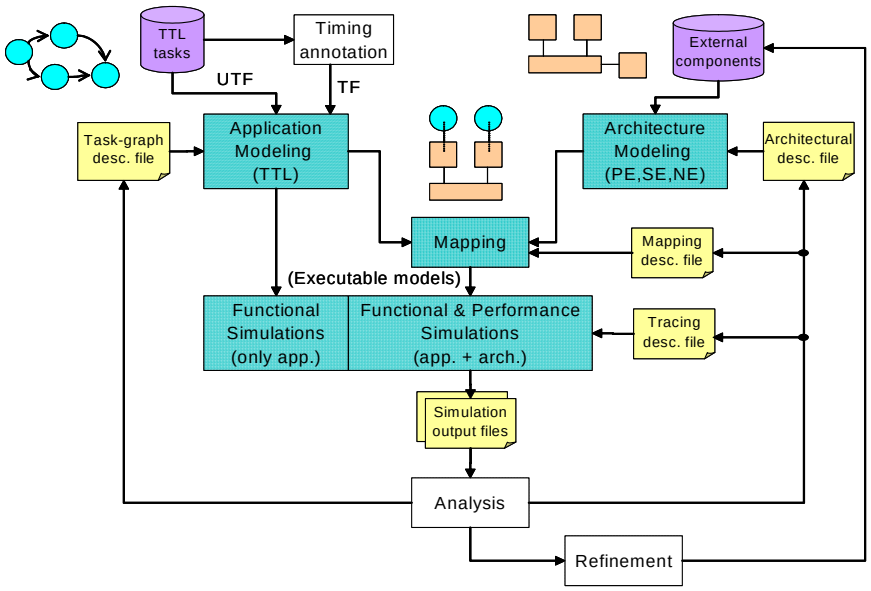


Figura 4.1: Flujo de diseño de *CASSE*

4.2.1 Modelado de la aplicación

CASSE sigue un modelo de programación basado en la interfaz *Task Transaction Level* (TTL) [22]. Según la especificación TTL, la aplicación es descrita como un proceso de red en la que las tareas paralelas se comunican entre si mediante canales unidireccionales.

En el entrono *CASSE*, las tareas de la aplicación son descritas en C/C++ y añadidas a *CASSE* como una librería externa. Sin embargo, la estructura del grafo de tareas (es decir, conexiones entre puertos y canales) y su configuración (es decir, tamaño del canal) son descritas en un fichero descriptivo de texto independientemente. La herramienta utiliza este fichero de descripción para instanciar y unir las tareas y canales creando un modelo de red ejecutable. Este modelo ejecutable es independiente de la arquitectura y puede ser simulado usando *CASSE* con el fin de validar la correcta funcionalidad de la aplicación y obtener información acerca de la comunicación y sincronización cargada en cada tarea/puerto que componen la aplicación, así como el máximo canal requerido.

4.2.2 Modelado arquitectural

CASSE proporciona un modelo sencillo y rápido de la arquitectura al describir el sistema como una composición modular de elementos predefinidos proporcionados por las librerías del sistema.

La librería de los elementos predefinidos está compuesta por:

- **Elementos de procesamiento** (PE): modelan unidades computacionales emulando su funcionalidad y temporización. Se puede mapear un número arbitrario de tareas en un solo PE, pero sólo una tarea puede estar activa en un instante determinado. Además, proporciona funcionalidad de sistema operativo a través de un planificador de tareas que soporta diferentes esquemas de arbitraje (prioridad, *round-robin*, TDMA), y de un controlador de interrupciones.

- **Elementos de almacenamiento** (SE): modelan elementos de memoria, como bancos de registros o memorias RAM estáticas. Permite emular el comportamiento de memorias de un solo puerto, dos puertos o multi-puertos.

- **Elementos de red** (NE): modelan interconexiones de bus compartido que incluyen un árbitro programable, decodificador de direcciones y *buffers* de entrada opcionales. Su principal función es la de interconectar los elementos de procesamiento y de almacenamiento.

Todos los elementos son configurables y pueden conectarse entre sí siguiendo una metodología *plug and play* mediante la interfaz ICCP (*Inter-Component Communication Protocol*). ICCP es un protocolo de comunicación abstracto que define una interfaz punto a punto y un grupo de primitivas de comunicaciones entre dos entidades, denominadas *Initiator* y *Target*. Tanto la interfaz ICCP como la librería de los elementos predefinidos se han desarrollado usando los estándares IEEE 1666 SystemC [23] y OSCI TLM [15]. Aparte de los elementos predefinidos, los modelos de arquitecturas pueden extenderse con nuevas funcionalidades mediante **componentes externos** (EC). Estos componentes externos, que pueden ser descritos en cualquier nivel de abstracción de SystemC, permiten extender la arquitectura de la plataforma con nuevas funcionalidades y pueden ser añadidos al modelo si sus interfaces son compatibles con ICCP.

Se emplea separadamente un fichero de descripción para especificar la composición de la arquitectura del sistema (número de elementos de cada tipo, número de interfaces por cada elemento, y sus interconexiones), así como su configuración (mapa de memoria, tamaños de las memorias, latencias de comunicación por interfaz, política del planificador de tareas, etc).

4.2.3 Mapeado y ejecución

Una importante contribución de la herramienta *CASSE* es que soporta el mapeado directo de las aplicaciones en los modelos de la arquitectura. Dicha

característica facilita la exploración de diferentes alternativas del espacio de diseño con mínimo esfuerzo.

Las tareas son mapeadas en los elementos de procesamiento y los canales en los elementos de almacenamiento. Varias tareas pueden ser mapeadas en un solo elemento de procesamiento, esto siendo posible al proporcionar la funcionalidad de un sistema operativo con un planificador de tareas con múltiples políticas. El resultado de la etapa de mapeado es un modelo ejecutable que contiene la aplicación/arquitectura seleccionada.

El entorno *CASSE* puede emplearse para realizar simulaciones funcionales de la aplicación modelada o bien simulaciones de la aplicación mapeada y ejecutada en el modelo de la arquitectura.

4.2.4 Análisis

Durante las simulaciones, la herramienta es capaz de obtener y almacenar información acerca del sistema en ejecución. Esta información permite al usuario analizar e identificar los cuellos de botella de la arquitectura, así como posibles optimizaciones del sistema a diferentes niveles. Basado en este análisis, se realizarán futuras iteraciones donde aplicación y arquitectura serán refinadas, o se seleccionará un nuevo mapeado.

4.2.5 Depurado

Una vez que el sistema está preparado para la implementación, los módulos *hardware* pueden perfeccionarse progresivamente desde descripciones en SystemC más abstractas a descripciones más precisas (incluso sintetizables). Dichos módulos pueden verificarse en el modelo de la plataforma simplemente reemplazando los elementos predefinidos por elementos externos que contienen el modelo preciso.

Capítulo 5

Herramienta CTAP

5.1 Introducción

En las primeras fases del flujo de diseño, un análisis del modelo a nivel TLM ayuda a tomar decisiones relacionadas con el posterior diseño a nivel RTL. El problema es que un modelo sin anotar no contiene información temporal necesaria para realizar dicho análisis. Por tanto, es necesario enriquecer dicho modelo con retardos temporales.

Este capítulo explica la funcionalidad de la herramienta CTAP, empleada para proporcionar dicha información temporal. En este Proyecto Fin de Carrera, esta herramienta se empleará para anotar los retardos de tiempo en el modelo *untimed CASSE Mixed-Level*.

5.2 Funcionalidad

CTAP es una herramienta que proporciona el número de ciclos de reloj que una declaración en C requiere una vez compilada. La herramienta provee esta información a través del cálculo del número de instrucciones a ensamblar necesarias para ejecutar dicha declaración, y proporciona el número de ciclos de reloj multiplicando dicho resultado con un determinado número de ciclos de reloj por instrucción. CTAP asume por defecto, que la ejecución de cada instrucción a ensamblar requiere de un ciclo de reloj. Sin embargo, el usuario puede especificar, si lo desea, una duración diferente para cada instrucción. Esta configuración es posible realizarla para todas las instrucciones de

forma global o por medio de una tabla en la que se establece un determinado número de ciclos de reloj para la ejecución de cada tipo de instrucción.

La herramienta CTAP proporciona un fichero de configuración en el que se establecen los ficheros que se desean aplicar a la herramienta, se especifica el compilador a emplear, y además, se establece el número de ciclos de reloj asignados a cada instrucción ensambladora. Una característica a destacar de la herramienta es que sólo trabaja con código descrito en lenguaje C.

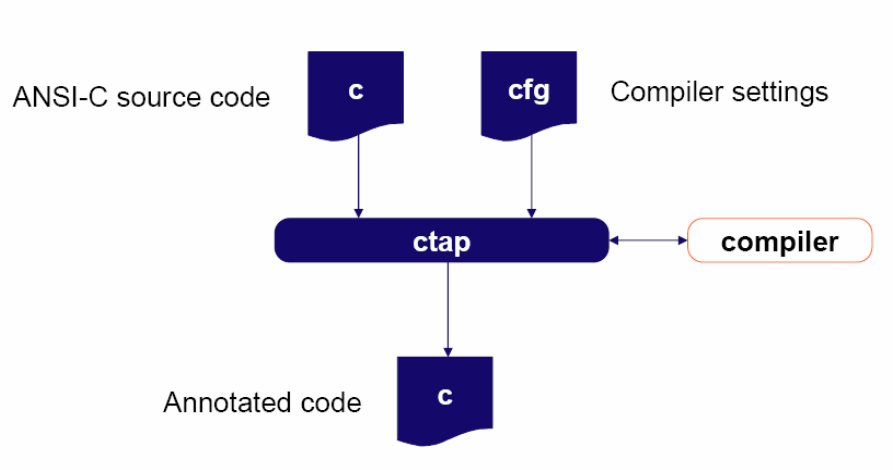


Figura 5.1: Proceso de la herramienta CTAP

El funcionamiento de la herramienta CTAP, representado en la *Figura 5.1* es simple; una vez que el fichero C está escrito, y configuradas las opciones del compilador, se aplica la herramienta. CTAP utiliza información del compilador para anotar en el código fuente la información temporal. Esta anotación es realizada en cada declaración de C indicando la duración estimada a través de funciones del tipo *duration(X)*, donde *X* corresponde al número de ciclos de reloj necesarios para ejecutar dicha declaración de C.

La *Figura 5.2* muestra un ejemplo del empleo de la herramienta CTAP. El código de la izquierda es el código original al que se le ha aplicado la herramienta, y el código de la derecha corresponde al fichero de salida donde ha sido anotada la información temporal. Obsérvese que el código anotado está basado en el código fuente original, añadiéndose el tiempo estimado para cada línea de código mediante funciones

duration(). Para este ejemplo, la herramienta ha sido configurada para igualar la latencia asociada a cada instrucción ensambladora a un ciclo de reloj (CPI=1).

```
//source code

1   int foo(int a, int b)
2   {
3   int x, c;
4   for (x = 0; x < 8; x++) {
5   if (a == 1)
6   c += b;
7   else
8   c -= b;}
9   return c;
10  }
```

```
//annotated code

1   int foo (int a, int b) {
2   duration(5);
3   int x, c;
4   duration(2);
5   for (x = 0; x < 8; x++) {
6   duration(2);
7   duration(2);
8   if (a == 1) {
9   duration(3);
10  c += b;}
11  else {
12  duration(1);
13  c -= b;}
14  }
15  duration(2);
16  return c;
17  }
```

Figura 5.2: Ejemplo de la información temporal proporcionada por CTAP

Capítulo 6

Aplicaciones

6.1 Introducción

Este capítulo presenta la aplicación utilizada como referencia lo largo de este Proyecto Fin de Carrera: el decodificador JPEG. Esta aplicación ha sido elegida para realizar la comparativa entre la plataforma *Mini-NoC* original y el modelo *CASSE Mixed-Level* desarrollado.

6.2 Aplicación JPEG

Este decodificador JPEG toma una imagen JPEG y la convierte a una imagen *bitmap* haciendo uso de un proceso de decodificación JPEG en su perfil básico [24]. El proceso de decodificación, como se puede apreciar en la *Figura 6.1*, ha sido dividido en tres tareas, cada una de las cuales se ha mapeado en un nodo *mMIPS* del *Mini-NoC*. De este modo se consigue establecer una aplicación multiprocesadora.

Así, cada uno de los pasos de la decodificación JPEG se ejecuta en un nodo separado de la plataforma *Mini-NoC*: el nodo 1 se encuentra en la posición (X, Y) = (0, 0), el nodo 2 corresponde a las coordenadas (1,0), el nodo 3 está en la posición (0,1), y el nodo (1,1) permanece sin utilizar.

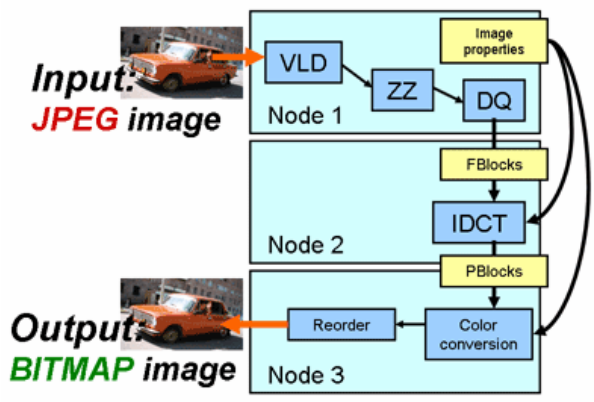


Figura 6.1: Proceso seguido por el decodificador JPEG

El decodificador toma una imagen comprimida de entrada sobre la que se aplican secuencialmente los siguientes procesos: decodificación de longitud variable [VLD], zigzag scan [ZZ], cuantización inversa [DQ], transformada discreta inversa del coseno [IDCT], conversión de color y reordenado. Finalmente, se obtiene la imagen reconstruida.

Una imagen puede ser separada en un número de bloques que son agrupados en macrobloques –MCU [Minimal Code Unit] – que describen una cierta región de la imagen. El proceso de crear un bloque y agruparlos en MCUs es realizado por el codificador. El decodificador encuentra esos MCUs en la imagen comprimida pudiendo entonces dividir un MCU en sus bloques representativos y decodificarlos. Los bloques amarillos (*Image properties*, *FBlocks*, *PBlocks*) de la *Figura 6.1* describen los datos que son enviados entre nodos (procesadores *mMIPS*).

Así, en esta aplicación la imagen de entrada es dividida en macrobloques en el nodo 1, que son enviados como bloques a través del *mNoC*. Los bloques son recibidos y procesados en el nodo 2 por la función IDCT y posteriormente reenviados al nodo 3, donde son procesados y reordenados en macrobloques. Por tanto, la ejecución de la aplicación es secuencial. Este hecho hace posible conocer cuándo es procesado y enviado un macrobloque desde el nodo 1, y cuándo es recibido y procesado en el nodo 3.

Capítulo 7

Modificaciones en la plataforma *Mini-NoC*

7.1 Introducción

Este capítulo describe la funcionalidad adicional que ha sido incorporada a la plataforma *Mini-NoC*, así como los cambios introducidos para optimizar las simulaciones. Comienza con una descripción de una nueva instrucción añadida al conjunto de instrucciones del procesador *mMIPS* con el objetivo de obtener el tiempo actual de la aplicación en una línea específica de código. El resto de cambios están enfocados a realizar un análisis óptimo de la comparativa. En primer lugar, se realizaron cambios en la librería *stdcomm* y en la aplicación JPEG, optimizando la simulación de la especificación a nivel RTL. En segundo lugar, se extendieron las memorias de los procesadores *mMIPS*, permitiendo una completa evaluación del modelo *CASSE Mixed-level*.

7.2 Instrucción *TIME_STAMP*

Como se introdujo en la sección 3.4, la plataforma *Mini-NoC* original permite obtener el tiempo total de simulación del sistema. Sin embargo, no es posible conocer el tiempo actual del sistema en una línea específica de la aplicación durante la simulación. Esto se debe a que el compilador utilizado en el procesador *mMIPS* del *Mini-NoC* es el

compilador C *LCC*, y las aplicaciones están descritas en el lenguaje C, en lugar de SystemC. Por tanto, obviamente éstas no son concientes de ninguna función que exista en la librería SystemC. Para solventar este problema es necesario añadir una instrucción ensambladora en el procesador *mMIPS* que permita obtener el tiempo actual en una línea específica del código de la aplicación.

Proporcionar al *Mini-NoC* original esta funcionalidad adicional conlleva cambiar, no sólo el procesador *mMIPS*, sino que además también requiere cambios en el compilador *LCC*. La razón es que la aplicación requiere ser compilada en el compilador *LCC* si desea ser ejecutada en la plataforma *Mini-NoC*. El fichero binario resultante de la compilación contiene el código ensamblador que es cargado en la memoria de instrucciones del *mMIPS*. Este circunstancia implica cambios en el compilador *LCC* y en el procesador.

Para aportar esta nueva funcionalidad se añadió una nueva instrucción al juego de instrucciones del *mMIPS* denominada *TIME_STAMP*. Esta instrucción puede ser añadida en cualquier parte de la aplicación donde se desee obtener el tiempo actual del sistema durante su simulación.

7.3 La nueva librería *stdcomm*

Como se adelantó en la sección 3.2.1, el procesador *mMIPS* es una versión simplificada de un procesador MIPS. Comparado con un procesador MIPS, presenta un juego de instrucciones reducido, lo que significa que algunas instrucciones requieren ser realizadas en *software*, como en su caso de los operadores módulo, multiplicación y división.

Las funciones *sc_send()* y *sc_receive()* implementadas en la librería *stdcomm* original incluyen la función módulo en varias declaraciones. Debido a que la función módulo es una de las instrucciones que no se encuentran en el juego de instrucciones de los procesadores *mMIPS*, cada vez que se llama a la función *sc_send()* y/o *sc_receive()* se crearán múltiples llamadas a funciones más simples soportadas por el procesador, lo que conlleva más código y mayor tiempo de ejecución. Además, el lugar en el que se

emplea la función modulo está localizada en un lugar relevante, ya que depende del tamaño del mensaje. Como resultado, conforme el tamaño del mensaje aumenta, también lo hace el tiempo de simulación requerido, llegando a una tendencia exponencial.

Con el fin de solventar este problema se ha modificado la librería *stdcomm*. Para ello, se ha sustituido el operador módulo por una declaración en C que realiza la misma funcionalidad, pero en este caso la función se realiza por hardware.

Como resultado, las primitivas de comunicación han sido optimizadas, y el tiempo de ejecución se vio considerablemente reducido. Por ejemplo, el tiempo de simulación empleado por la aplicación JPEG en la decodificación de una imagen de 32x24 pixels fue reducido, de 8 horas, a 48 minutos.

7.4 Decodificador JPEG

Con el fin de realizar las métricas en el decodificador JPEG durante la comparativa de ambas plataformas, la opción *verbose* de la aplicación fue desactivada. Esta opción permite al usuario observar los pasos seguidos por el decodificador durante la ejecución a partir del uso funciones *mprintf()* (ver sección 2.3.2 de la Memoria) que son distribuidas a lo largo de la aplicación y cuya realización requiere de un número de ciclos de reloj significativo. Este número de ciclos es causado por los continuos accesos a memoria, e introduce una incorrección en la comparativa. En consecuencia, esta información fue eliminada por considerarse no relevante para la ejecución de la aplicación.

7.5 Extensión de memoria

La plataforma *Mini-NoC* reserva una zona de memoria RAM para el almacenamiento de datos de usuario. Estos 3.5kbytes de memoria están destinados a almacenar los datos usados por la aplicación. Este tamaño de memoria no permite almacenar imágenes de entrada ni de salida de mayor tamaño.

Para permitir almacenar imágenes de entrada mayores, el mapa de memoria del primer nodo fue modificado con el fin de aumentar el espacio de memoria destinado a este fin. Además, las memorias del resto de nodos también fueron modificadas para permitir el almacenamiento de datos intermedios y de la imagen de salida.

Capítulo 8

Modelo *CASSE Mixed-Level*

8.1 Introducción

En este capítulo se presenta al modelo *CASSE Mixed-Level*, así como los pasos seguidos en su implementación: modelado de aplicación, modelado de la arquitectura, mapeado y anotaciones temporales.

Cada sección del capítulo está destinada a cada uno de estos pasos. La primera presenta el modelado de la aplicación, en la que se describe cómo fueron modeladas las aplicaciones en el modelo *CASSE Mixed-Level*. Seguidamente, se detallan las diferentes etapas seguidas para implementar la arquitectura, así como los módulos que la componen. En la tercera sección, correspondiente el proceso de mapeado, se explica cómo se mapeó el decodificador JPEG sobre el modelo. Finalmente, el capítulo termina con una sección en la que se detalla el procedimiento seguido para obtener la información temporal, y anotar el modelo *untimed*.

8.2 Modelado de la aplicación

Para modelar una aplicación en la herramienta *CASSE*, las tareas que componen la aplicación deben ser compatibles con la interfaz de tareas de *CASSE*. De acuerdo con esto, las tareas deben de ser introducidas dentro de una clase de C++, y además, deben de heredar la clase *task_if* de la herramienta.

La plataforma *Mini-NoC* utiliza el paso de mensajes para comunicarse entre procesadores, por lo que resulta necesario adaptar las primitivas de comunicación para permitir la compatibilidad con los protocolos de comunicación de la herramienta *CASSE*. Como cualquier otra aplicación a modelar en la herramienta, las primitivas de comunicación también deben de cumplir con la interfaz de tareas de *CASSE* (*task_if*). Por tanto, la librería *comm* - que implementa las primitivas- debe heredar todos los atributos y funciones de la interfaz de tareas de *CASSE* (*task_if*).

Por otro lado, las tareas de la aplicación requieren de las primitivas de comunicación para establecer comunicaciones con el resto de nodos. Por tanto, para que las tareas puedan comunicarse necesitan heredar estas funciones de la librería *comm*.

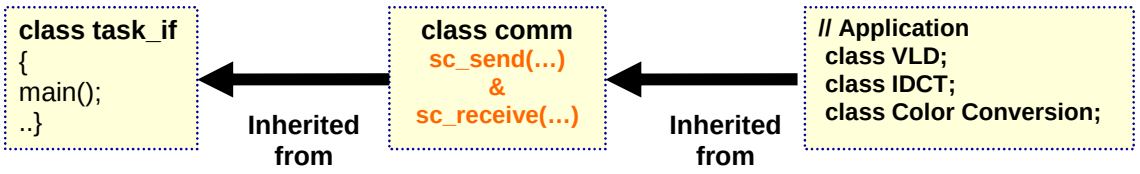


Figura 8.1: Jerarquía de clases

La jerarquía de clases propuesta se ilustra en la *Figura 8.1*. La clase base corresponde con la clase *task_if*, y la clase derivada es la librería *comm*. De esta forma, la librería *comm* hereda todas la características de la clase base, pero además, añade sus propias funciones *sc_send()* y *sc_receive()*. En la misma línea, se propone que las clases de la aplicación sean clases derivadas de la clase *comm*, heredando así las capacidades de la clase base y también las de la clase *comm*. Esta jerarquía permite a la aplicación cumplir con la interfaz de tareas de *CASSE*, además de acceder a las funciones *sc_send()* y *sc_receive()*.

Obsérvese que mientras la clase base no se ha modificado en el proceso, la librería *comm* sí que ha requerido ser modificada en el presente Proyecto Fin de Carrera. Esto se debe a que ésta depende directamente de la plataforma, aunque la ventaja es que una vez modelada, puede emplearse en cualquier aplicación. Por otro lado, la aplicación sí que funciona al margen de la plataforma, no siendo consciente en ningún momento de dónde está siendo mapeada.

8.2.1 *Comm* class

La clase *comm* es una nueva librería que implementa las primitivas de comunicaciones del modelo *Mixed-Level*: las funciones *sc_send()* y *sc_receive()*. Estas funciones están basadas en las originales de la plataforma *Mini-NoC*, implementadas en la librería *stdcomm*, que controlan las comunicaciones con la red a partir de directivas *load* y *store* a posiciones de memoria.

Como se explicó en la sección 4.2.2, ICCP es un protocolo de comunicación abstracto que define una interfaz TLM punto-a-punto, y un grupo de primitivas entre dos entidades denominadas *Initiator* y *Target*. La funcionalidad del protocolo ICCP es la de interconectar todos los componentes de la arquitectura utilizados en *CASSE*, y permitir la comunicación entre ellos. La interfaz ICCP proporciona dos métodos básicos para la comunicación entre una entidad *Initiator* y una entidad *Target*: *readBurst()* y *writeBurst()*.

Debido a que el modelo *Mixed-Level* se comunica con la red a nivel de transacción, las primitivas de comunicación han sido modificadas ligeramente para adaptarlas al protocolo ICCP de *CASSE*. La adaptación de las primitivas requirió de la sustitución de las directivas *load*/*store* por *write/read*.

La implementación de la función *sc_send()* de la librería *comm* está basada en la funcionalidad de la función original de la librería *stdcomm* de la plataforma *Mini-NoC*. Esta primitiva divide el mensaje en palabras de 4 bytes, y cada una de ellas se envía mediante la función *sc_send_word()*. Con el fin de adaptarlas al protocolo ICCP de *CASSE*, se añadirán los métodos de comunicación *readBurst()* y *writeBurst()* a la función *sc_send_word()*, los cuales, a partir de la entidad *Initiator*, realizan las transacciones con la red, como se refleja en las líneas 4 y 8 de la *Figura 8.2*.

La funcionalidad de la primitiva *sc_receive()* de la librería *comm* también está basada en la original, que se encarga de recibir todos los paquetes y reconstruir el mensaje. Cada palabra es leída de la red a través de llamadas a la primitiva

sc_receive_word(). La función primeramente chequea si el dato está disponible, y en caso afirmativo lo lee. Ya que es en dicha función *sc_receive_word()* donde se realizan las comunicaciones con la red, en ella se añaden los métodos *readBurst()* y *writeBurst()* de la interfaz ICCP para su adaptación con el protocolo ICCP.

```
1   int comm::sc_send_word(const int *ctrlword, const int *data,
    int try_count){
2      InitiatorPort* InitP = VML->getInitiatorP(0);
3      …
4      if(ReadBurst(InitP,0x80000004,4,&aux) == ICCP_ERROR){..}
5
6      while(!(aux & SC_BIT_SEND_READY));
7      aux = *data;
8      WriteBurst(InitP,0x80000000,4,&aux);
9      aux = *ctrlword | SC_BIT_SEND;
10     WriteBurst(InitP,0x80000004,4,&aux);
11     …}
```

Figura 8.2: Extracto de código de la primitiva *sc_send()* de la clase *comm*

8.2.2 Aplicación Gossip

Gossip es una aplicación sencilla escrita en C++, desarrollada para comprobar la comunicación entre los nodos a través de la red, verificando así la correcta funcionalidad de la clase *comm*, y de la arquitectura del modelo *Mixed-Level*. La aplicación está compuesta por cuarto tareas que envían y reciben un mensaje por la red a parir de llamadas a las primitivas de comunicación *sc_send()* y *sc_receive()*.

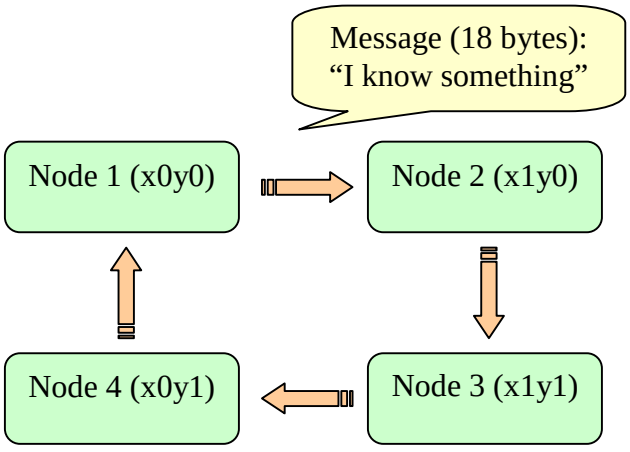


Figura 8.3: Aplicación *Gossip*

El primer nodo envía el mensaje "*I know something!*" al segundo nodo, y éste lo reenvía al tercero. La aplicación continúa hasta que el primer nodo recibe el mensaje que él mismo envió, con lo que queda comprobada la comunicación a través de la red. La función de esta aplicación se muestra en la *Figura 8.3.*

Las cuatro tareas son creadas en un fichero de descripción denominado *taskgraph.txt*. Este fichero se emplea en la herramienta para describir la estructura del grafo de tareas y establecer su configuración.

8.2.3 Modificaciones en el decodificador JPEG

Esta sección tiene como objetivo explicar las modificaciones realizadas sobre la aplicación del decodificador JPEG, introducida en el *Capitulo 6*, para adaptarla al entorno de *CASSE*.

Debido a que esta aplicación está descrita en C, la primera modificación fue trasladar las tres tareas a clases de C++. Seguidamente, como se explicó al comienzo de la sección 8.2, fue necesario definir las tareas que componen la aplicación como clases derivadas de la clase *comm* para modelar la aplicación en la herramienta *CASSE*.

Finalmente, se retiró de la aplicación código específicamente destinado para la plataforma *Mini-NoC*, como en el caso de las llamadas a funciones específicas del procesador *mMIPS mt_halt()* y *mprintf()*, empleadas para el depurado de código C e implementadas en la librería *mtools*.

Una vez que las modificaciones se han realizado, las tres tareas se instanciaron en el fichero de descripción *taskgraph.txt*, como se representa en la *Figura 8.4*.

```
1  # Process Network file #
2  # Tasks creation #
3  .CREATE -TASK step1 -N_PORT 0 ;
4  .CREATE -TASK step2 -N_PORT 0 ;
5  .CREATE -TASK step3 -N_PORT 0 ;
6  # eof #
```

Figura 8.4: Fichero de descripción *taskgraph.txt*

Obsérvese que en la *Figura 8.4* se le asigna a cada tarea un número de puertos. Esta asignación es necesaria ya que la herramienta *CASSE* se fundamenta en un modelo de programación paralelo basado en la interfaz TTL. La especificación TTL describe la aplicación como una red de procesos donde las tareas paralelas se comunican entre si a través de canales unidireccionales. El modelo *Mixed-Level* establece la comunicación a través del paso de mensajes, por lo que las tareas no necesitan comunicarse mediante canales y, en consecuencia, no son necesarios los puertos. Sin embargo, en el fichero de descripción de la herramienta es condición indispensable establecer un número de puertos para la instanciación de las tareas, aunque éste sea nulo.

8.3 Modelado de la arquitectura

El modelo *Mixed-Level* está compuesto por dos niveles de abstracción: TLM (*Transaction Level Modeling)* y RTL (*Register Transfer Level*). Los cuatro nodos están descritos a nivel TLM, mientras que el *NOC* es un componente descrito a nivel RTL debido a que incluye la red *mNoC* de la plataforma *Mini-NoC, c*omo se representa esquemáticamente en la *Figura 8.5*.

Esta sección explica cada uno de los pasos seguidos para obtener el modelo desarrollado en este Proyecto Fin de Carrera. Así, se comienza describiendo el proceso de creación de los nodos del modelo, explicando posteriormente la adaptación e integración de la red al modelo.

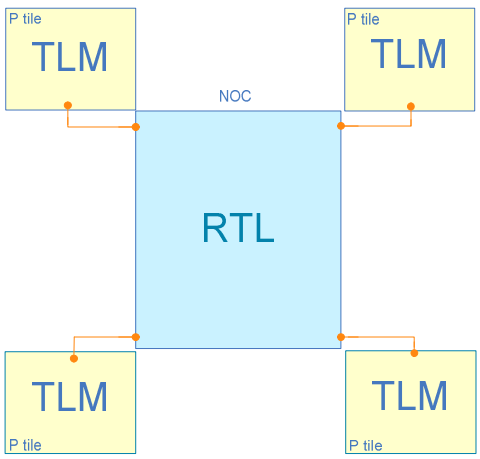


Figura 8.5: Modelo *Mixed-Level*

8.3.1 Creando los nodos de la plataforma *Mini-NoC* en la herramienta *CASSE*

El objetivo del primer paso es crear los cuatro nodos de la plataforma *Mini-NoC* del nuevo modelo en la herramienta *CASSE*. Para abordar este paso, los cuatro nodos fueron modelados con elementos predefinidos de *CASSE* proporcionados por las librerías de la herramienta. Los procesadores *mMIPS* fueron modelados con elementos de procesamiento (PE), las memorias locales con elementos de almacenamiento (SE) y los módulos *MEMDEV* con elementos de red (NE), como se representa en la *Figura 8.6*.

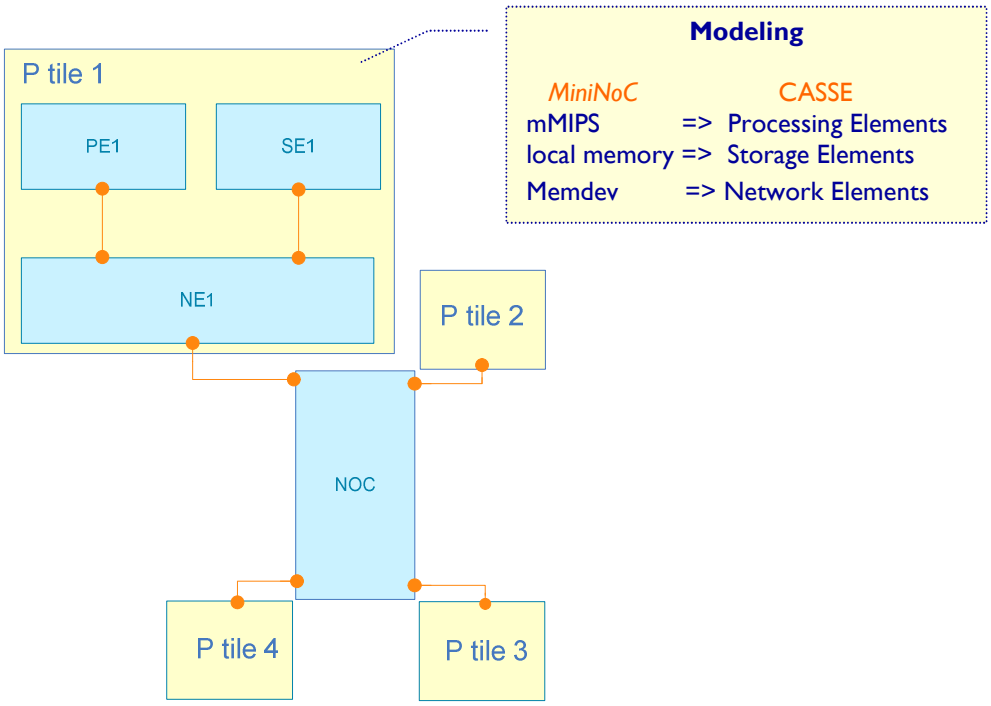


Figura 8.6: Modelado de los nodos procesadores

8.3.2 Adaptando e integrando el componente *mNoC*

Una vez que los nodos han sido modelados por elementos predefinidos de la herramienta *CASSE*, el siguiente paso es la de adaptar e integrar los componentes *mNoC* de la plataforma *Mini-NoC* en el modelo.

Para integrar el componente *mNoC*, se ha introducido un elemento externo (EC) de *CASSE* en el modelo. Este modelo está compuesto por los cuatro *routers* e interfaces de red del *Mini-NoC* original descritos a nivel RTL.

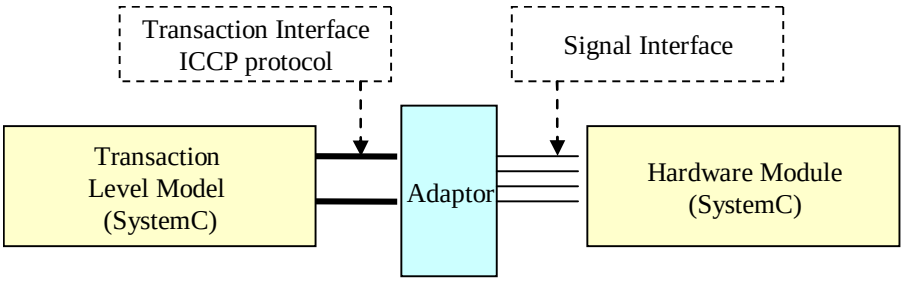


Figura 8.7: Los adaptadores conectan modelos de distintos niveles de abstracción

Ahora que el módulo *NOC* está integrado, el siguiente paso conectarlo a los cuatro nodos del modelo. Debido a que la red está descrita a nivel RTL, no cumple con el protocolo ICCP, por lo que resultó necesario añadir módulos adaptadores para llevar a cabo las transacciones.

Estos módulos se añaden con el fin de adaptar los niveles de abstracción, convirtiendo las señales de la interfaz de red a nivel de transacción. Esta característica permite que parte del diseño pueda ser simulado a un nivel de transacción, mientras que otras partes del diseño pueden ser simuladas a nivel *hardware*, como se representa en la *Figura 8.7*.

Por tanto, con el fin de adaptar los elementos de la plataforma *Mini-NoC* a la interfaz ICCP, se añadieron los adaptadores al componente externo, y como resultado, el *NOC* tiene la capacidad de realizar transacciones con los nodos.

8.3.2.1 Módulo adaptador

El adaptador es el módulo que permite la adaptación de los componentes de la plataforma *Mini-NoC* al resto del modelo *Mixed-Level*. Este módulo ha sido implementado mediante un puerto *Target* donde las transacciones son llevadas a cabo a través del protocolo ICCP. Unido al puerto *Target* hay un *slave*, que implementa las

funciones *read()* y *write().* Estas funciones son las primitivas a las que se llama cuando se llevan a cabo las transacciones en el protocolo ICCP.

Por otro lado, en este módulo se añade un hilo de ejecución (*thread*), denominado *exec().* Este hilo accede a las señales de la interfaz de red y se encarga de gestionar el protocolo de intercambio necesario para llevar a cabo el envío y recepción de paquetes a través de la red. Para este propósito, se requiere de una comunicación entre ambos procesos, *exec()* y *slave,* realizada mediante *buffers* donde se almacenan los datos y que permiten la comunicación entre ambos niveles de abstracción.

La *Figura 8.8* ilustra la implementación del adaptador y uno de los nodos del modelo. La transacción se inicia con la llamada a los métodos *writeburst()/readburst()* de la interfaz ICCP, cuando las primitivas de comunicación *sc_send()* y/o *sc_receive()* son llamadas en una tarea mapeada en un elemento de procesamiento (PE). Estas transacciones se llevan a cabo en el módulo adaptador llamando a las funciones *write*/*read* implementadas en el *slave*, el cual se comunica a su vez con el hilo de ejecución que gestiona las señales apropiadas de acuerdo con el tipo de transacción solicitada.

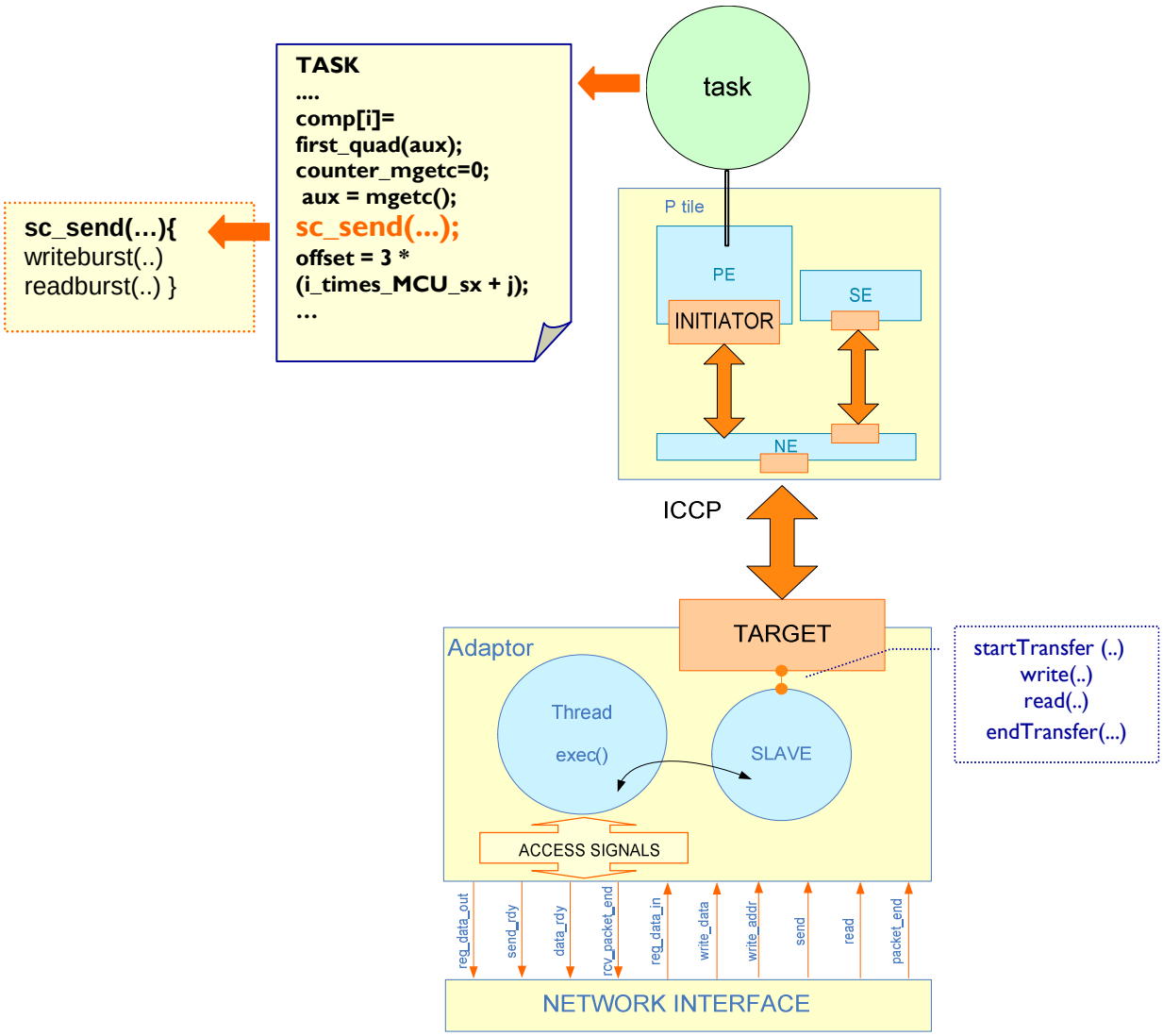


Figura 8.8: Implementación del módulo adaptador

8.3.2.2 Componente *NOC*

Como se expuso previamente, el componente *NOC* de la plataforma modelada contiene los componentes originales de la red *mNOC* de la plataforma *Mini-NoC*: los cuatro routers y sus correspondientes cuatro interfaces de red. La integración de dichos componentes se ha realizado mediante el fichero *NINET.h* (*Network Interfaces and NETwork*), donde se han realizado cuatro instancias a la interfaz de red y una a la *network2x2.h*, que ya incluye a la conexión de los cuatro *routers* (*Figura 8.9*). Finalmente, al módulo *NOC* se le conectaron los cuatro adaptadores en el fichero

NINET.h, obteniendo así el módulo *NOC* deseado, representado esquemáticamente en la *Figura 8.10.*

```
1   // ninet.h file
2   #include <systemc.h>
3   #include "network2x2.h"
4   SC_MODULE(NINET){
5   // PORTS
6      sc_in< bool > clk;
7      sc_in< bool > rst;
8      sc_in< sc_bv<32> > nix0y0_reg_data_in;
9      sc_in< bool > nix0y0_write_data;
10     …
11  // Pointer declarations
12     NETWORK_INTERFACE *nix0y0;
13     NETWORK_INTERFACE *nix0y1;
14     …
15     NETWORK2x2 *net;
16     SC_CTOR(NINET){
17     //Objects instanciated
18        nix0y0 = new NETWORK_INTERFACE("nix0y0");
19        nix0y1 = new NETWORK_INTERFACE("nix0y1");
20        …
21        net = new NETWORK2x2("net");
22     // Mapping signals to the objects ports
23        net->clk(clk);
24        net->rst(rst);
25        net->x0y0dout(nix0y0_data_in);
26        net->x0y0req_dp(nix0y0_req_in);
27  …}};
```

Figura 8.9: Extracto del código del fichero *Ninet.h*

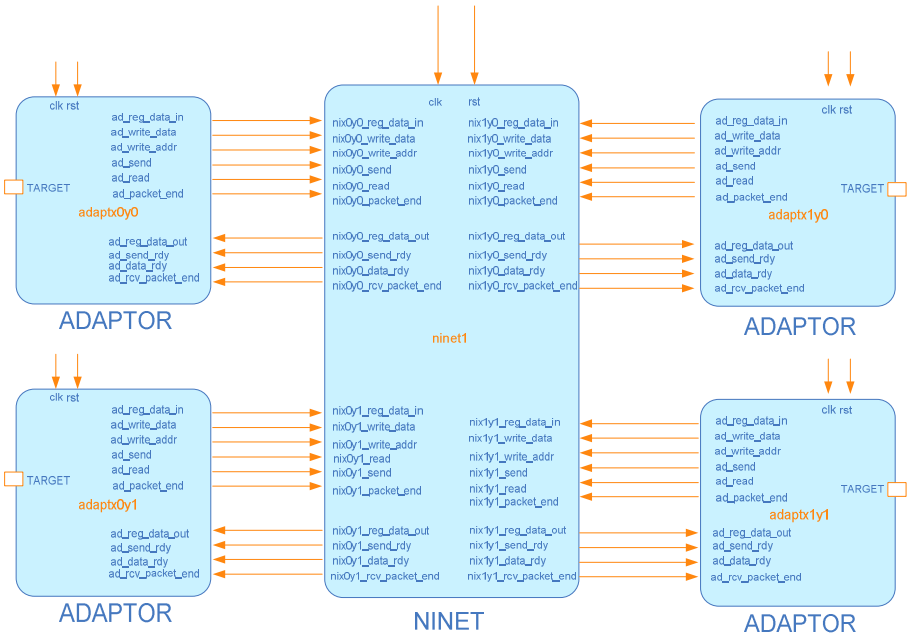


Figura 8.10: Conexiones del componente *NOC*

8.3.3 Modelo *CASSE Mixed-Level*

El modelo se completó una vez que el componente *NOC* fue añadido al modelo inicial. Como resultado, la arquitectura de la plataforma está compuesta por cuatro nodos – que contienen la misma funcionalidad que los nodos de la plataforma *Mini-NoC*- y de un elemento *NOC* – que adapta e integra el componente *mNOC* en *CASSE*.

Cada nodo está implementado con elementos predefinidos de *CASSE*: PE, SE, y NE. El componente *NOC* está compuesto por los *routers* e interfaces de red (componentes *mNOC*), añadiéndose los adaptadores con el fin de adaptarlos al protocolo ICCP.

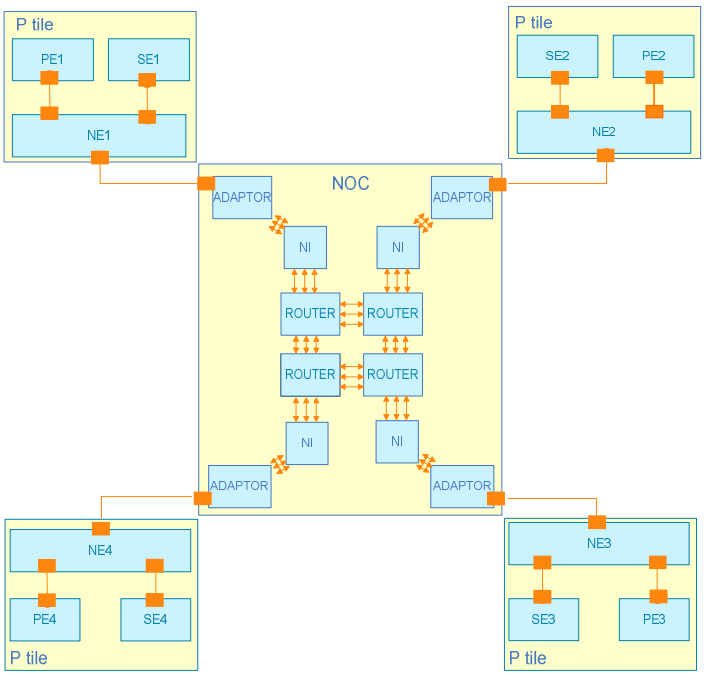


Figura 8.11: Modelo *CASSE Mixed-Level*

Cada uno de los módulos del modelo fueron creados y configurados mediante el fichero de descripción denominado *architecture.txt*. Este fichero se emplea en la herramienta para especificar la composición de la arquitectura del sistema. Para especificar todo el sistema fueron necesarias únicamente 100 líneas de código. Cada nodo requirió cerca de 25 líneas, en las que los elementos predefinidos y el elemento

externo fueron creados y configurados. Cada elemento (procesamiento, almacenamiento, red y externo) es creado indicando el número de interfaces e interconexiones. También se configura el mapa de memoria, tamaños de memorias, latencias de comunicación por cada interfaz, política del planificador de tareas, etc. El reloj también está implementado en este fichero, y coincide con el período de reloj del *Mini-NoC* (10ns). Las líneas utilizadas para crear y configurar el *Tile 1* (nodo 1) se muestran en la *Figura 8.12*.

```
1   # architectural file #
2   .CREATE -CLOCK clock1 -PERIOD 10 -UNIT SC_NS ;
3   .CREATE -EXTERNAL noc ;
4   # TILE 1 #
5   .CREATE   -PROCESSING proc1 -N_INIT 1 ;
6   .CONFIGURA      -PROCESSING proc1 -INIT 0 -WIDTH 32 -LAT 0 0 0 0 -
    CONNID 0 ;
7   .CONFIGURA      -PROCESSING proc1 -TS MLQ 100 50 0 ;
8   .CONFIGURA      -PROCESSING proc1 -PTW YIELD 5 2 1 5 ;
9   .CONFIGURA      -PROCESSING proc1 -INIT 0 -MEMORYMAP 0x00000000
    0x000fffff mem1 ;
10  .CREATE   -STORAGE     mem1 -N_TARGET 1 ;
11  .CONFIGURA      -STORAGE     mem1 -SIZE 1048576 ;
12  .CONFIGURA      -STORAGE     mem1 -TARGET 0 -WIDTH 32 -LAT 0 0 0 ;
13  .CREATE   -NETWORK     bus1 -N_INPUT 1 -N_OUTPUT 2 ;
14  .CONFIGURA      -NETWORK     bus1 -ARBITER ROUNDROBIN ;
15  .CONFIGURA      -NETWORK     bus1 -WIDTH 32 -BUFFERED n -I_LAT 0 0 0 -
    O_LAT 0 0 0 0 ;
16  .CONFIGURA      -NETWORK     bus1 -OUTPUT   0 -RANGE 0x00000000
    0x000fffff ;
17  .CONFIGURA      -NETWORK     bus1 -OUTPUT   1 -RANGE 0x80000000
    0xffffffff ;
18  .CREATE -LINK LPB1 -WIDTH 32 ;
19  .CREATE -LINK LBM1 -WIDTH 32 ;
20  .CREATE -LINK LECB1 -WIDTH 32 ;
21  .BIND -CLOCK clock1 TO -PROCESSING proc1 ;
22  .BIND -CLOCK clock1 TO -STORAGE mem1 -TARGET 0 ;
23  .BIND -CLOCK clock1 TO -NETWORK bus1 -INPUT ;
24  .BIND -CLOCK clock1 TO -NETWORK bus1 -OUTPUT ;
25  .BIND -LINK LPB1 TO -PROCESSING proc1 -INIT 0 ;
26  .BIND -LINK LPB1 TO -NETWORK bus1 -INPUT 0 ;
27  .BIND -LINK LBM1 TO -STORAGE mem1 -TARGET 0 ;
28  .BIND -LINK LBM1 TO -NETWORK bus1 -OUTPUT 0 ;
29  .BIND -LINK LECB1 TO -NETWORK bus1 -OUTPUT 1 ;
30  .BIND -LINK LECB1 TO -EXTERNAL noc -TARGET 0 ;
31
32  // configuration lines of tiles 2, 3, 4 omitted
```

Figura 8.12: Extracto del fichero de descripción *architecture.txt*

8.4 Mapeado

Ahora que la plataforma está ya desarrollada, el siguiente paso es mapear las tareas que componen la aplicación JPEG en el modelo.

Como se adelantó en la sección 6.2, el decodificador JPEG está dividido en tres tareas que son mapeadas en tres nodos. Cada paso se ejecuta en un nodo separado: el nodo 1 se encuentra en (X, Y) = (0, 0), el nodo 2 está en (1,0), el nodo 3 está en (0,1), y el nodo (1,1) permanece sin utilizar.

Debido a que *CASSE* ya permite el mapeo directo de la aplicación sobre la arquitectura, el procedimiento de mapeado se describe rápidamente mediante un fichero de descripción denominado *mapping.txt* representado en la *Figura 8.13*. El mapeado de las tareas sólo requirió de 3 líneas de código, cada una de ellas se encarga de mapear cada tarea en un procesador.

```
1   # mapping file #
2   .MAP -TASK step1 INTO -PROCESSING proc1 ;    # x0y0 #
3   .MAP -TASK step2  INTO -PROCESSING proc2 ;   # x1y0 #
4   .MAP -TASK step3  INTO -PROCESSING proc4 ;   # x0y1 #
5   # eof #
```

Figura 8.13: Fichero de descripción mapping.txt

8.5 Anotaciones temporales y análisis

El modelo anotado es una aproximación del sistema en la que los retardos temporales son insertados en el modelo *untimed*. Estas anotaciones temporales están compuestas por información temporal extraída a nivel de microarquitectura, la cual se realiza mediante declaraciones del tipo *wait(sc_time)* relacionadas con el tiempo computacional de una funcionalidad específica.

8.5.1 Alternativas para las anotaciones temporales

Con el fin de realizar la comparativa es necesario anotar el modelo *untimed* de *CASSE* con información temporal obtenida de la especificación RTL. Para abordar este aspecto, se han propuesto diferentes alternativas descritas en detalle en la sección 7.5.1 de la Memoria. En este Resumen sólo se presentan las opciones elegidas:

Opción 2:

Consiste en una estimación del tiempo de ejecución de bloques de código C de la aplicación. El número de instrucciones ensambladoras generadas por el compilador de cada línea de código se obtiene multiplicando el número de instrucciones ensambladoras por una estimación del número de ciclos de reloj por instrucción ensambladora. Con la información temporal de cada declaración es posible obtener el número de ciclos de reloj de cada bloque de código de la aplicación añadiendo contadores. Las medidas obtenidas se anotan a continuación en el modelo de *CASSE*. Observe que la ventaja de esta opción es que sólo requiere de la compilación de la aplicación.

Opción 5:

Instrumentar la especificación RTL para medir el número exacto de ciclos de reloj que requieren partes largas de código de la aplicación. Utilizar esta estimación para anotarla el código de la aplicación en el modelo de *CASSE*. Esta aproximación requiere de la compilación y ejecución de toda la especificación RTL, y al estimar largas partes de código, puede resultar impreciso.

De acuerdo con las opciones propuestas, se decidió explorar la opción 2 y la opción 5. La primera es la más probable desde un contexto de diseño real y además es la más interesante desde un punto de vista de diseño y herramienta. La opción 5 se ha elegido ya que proporciona una manera de considerar la opción de instrumentar el sistema a nivel RTL sin dedicarle mucho esfuerzo. Finalmente, no hemos considerado las opciones 3 y 4, ya que requieren mucho tiempo y la precisión añadida al modelo es cuestionable.

Con el fin de decidir la mejor opción se evaluará la precisión obtenida en ambas opciones. Para abordar esta evaluación, se han obtenido medidas a partir de la decodificación de la imagen *surfer* de 32x24 pixels, que se representa a tamano real en la *Figura 8.14*). Para la obtención de las anotaciones temporales de la opción 2 se ha propuesto la herramienta CTAP.

Figura 8.14: Imagen de entrada *surfer.jpg* (tamaño real)

8.5.2 Anotaciones temporales en *CASSE*

El entorno *CASSE* proporciona dos funciones para anotar información temporal: *DELAY_CYCLES(ncycles)* y *DELAY_TIME(period,time_unit)*, cuyas implementaciones están basadas en la función *wait(sc_time)* de la librería SystemC. La diferencia entre ambas funciones es que en la primera es posible asignar el número de ciclos de reloj, mientras que en la segunda, el tiempo puede asignarse directamente indicando el valor y la unidad de tiempo utilizada.

Ambas han sido utilizadas en las dos opciones contempladas para el modelo *Mixed-Level timed*. La función *DELAY_TIME()* se empleó para anotar los retardos proporcionados a través de la instrumentación RTL (opción 5), ya el tiempo se provee en nanosegundos. Por otro lado, la función *DELAY_CYCLES()* se utilizó para anotar la información temporal proporcionada por la herramienta CTAP, ya que está la facilita en ciclos (opción 2).

8.5.3 Instrumentación del *mMIPS*

El modelo anotado basado en la instrumentación del *mMIPS* consiste en la instrumentación de la especificación RTL para medir el número exacto de ciclos de reloj de largas partes de código C. El número de ciclos de reloj pueden ser obtenidos utilizando las funciones de etiquetas temporales o *time stamp* distribuidas en la aplicación JPEG en su ejecución en la plataforma *Mini-NoC*. Con la información resultante, es posible anotar el código en el modelo *CASSE Mixed-Level*.

Para seguir el proceso realizado para anotar la información temporal, se ha aportado como ejemplo la función IDCT del nodo 2, cuyo extracto se muestra en la *Figura 8.15*. Primeramente, las funciones *time stamp* fueron añadidas antes y después de la llamada a la función IDCT mediante la instrucción *TIME_STAMP*. Seguidamente, la aplicación JPEG fue ejecutada en el simulador de SystemC para el *Mini-NoC* como se explicó en la sección 3.4. Cuando la aplicación se ejecuta, muestra en la salida estándar la información temporal. El número total de ciclos de reloj que se necesitaron en la ejecución de la IDCT fue obtenido mediante la substracción del valor de ambos *time stamps*. Posteriormente, se realizó una media de todas las llamadas a la función durante la ejecución. Finalmente, la media obtenida fue anotada en el modelo *Mixed-Level* mediante la función *DELAY_TIME(),* tal y como se muestra en la línea 7 de la *Figura 8.15*. Este proceso fue seguido en las tres tareas que componen la aplicación.

```
1  //STEP 2:       IDCT
2   …
3  while(blocks_remain){
4  sc_receive(&input, sizeof(FBlock));
5  IDCT(&input, &output);
6
7  DELAY_TIME(1141570,SC_NS);
8
9  sc_send(ADDR_STEP2TO3, &output, sizeof(PBlock));
10 sc_receive(&blocks_remain, sizeof(int));}
11 …
```

Figura 8.15: Ejemplo de la anotación realizada mediante la instrumentación del *mMIPS*

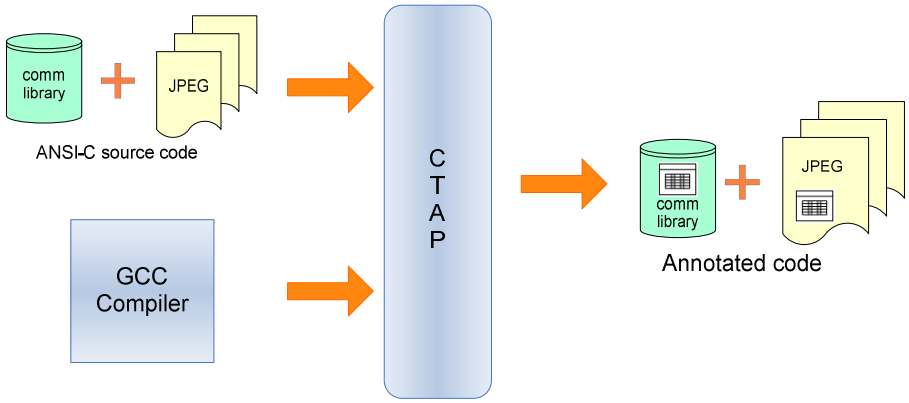
Un inconveniente de esta anotación es que las prestaciones del modelo anotado resultante sólo pueden ser analizadas para la misma imagen en la que la información temporal ha sido obtenida (*surfer.jpg*). Por consiguiente, al ser dependiente de los datos, no es posible analizar las prestaciones del modelo para diferentes imágenes, limitando la comparativa entre plataformas a una sola imagen. Otra desventaja es que requiere compilar y ejecutar la aplicación para obtener dicha información temporal.

8.5.4 Empleando la herramienta CTAP

Como se explicó en la sección 5.2, CTAP proporciona el número de ciclos de reloj que una declaración en C necesita, una vez compilada. Esta información se obtiene del compilador empleado por la herramienta y del número de ciclos de reloj por instrucción ensambladora establecido en el fichero de configuración. Una vez que la herramienta se ha aplicado, el resultado es el fichero fuente anotado con la información temporal.

El compilador *LCC* del *mMIPS* no puede aplicarse a la herramienta CTAP. El problema de utilizar dicho compilador en CTAP es que no soporta el formato estándar STABS de depurado, lo que hace imposible encontrar la relación entre las instrucciones ensambladoras y las líneas de código en C. Ya que no es posible usar el compilador *LCC* y el compilador *GCC* se ajustaba perfectamente a la herramienta, se utilizó éste en su lugar.

El modelo anotado ha sido obtenido aplicando la herramienta CTAP. En primer lugar, esta herramienta se aplicó el decodificador JPEG y en la librería *comm*. El compilador utilizado en CTAP fue el compilador *GCC* y la herramienta fue configurada con la opción por defecto, en la que se asume un ciclo de reloj por cada instrucción ensambladora. Como resultado, el código fue anotado con la información temporal de cada declaración de C. El siguiente paso consistió en anotar esa información temporal en la aplicación JPEG y en la librería *comm* del modelo *CASSE untimed*. Las partes secuenciales del código fueron anotadas mediante contadores y el resto fue anotado en cada declaración de C.



Fgura 8.16: Proceso seguido para aplicar CTAP en el decodificador JPEG

En la *Figura 8.17* se presenta parte del código de la función IDCT como ejemplo de cómo se anotó el código en cada línea a través de la función *DELAY_CYCLES()*.

```
1   //STEP 2:        IDCT
2   #define CMUL(C, x)  (((C) * (x) + (1 << (C_BITS-1))) >> C_BITS)
3   ..
4   DELAY_CYCLES(k*34);
5   z1[2] = SUB(CMUL( 8867, Y[2]), CMUL(21407, Y[6]));
6   DELAY_CYCLES(k*34);
7   z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
8   DELAY_CYCLES(k*3);
9   z3[1] = z2[1];
10  DELAY_CYCLES(k*3);
11  z3[2] = z2[2];
12  ...
```

Figura 8.17: Ejemplo de la anotación realizada mediante CTAP

Al configurarse la herramienta CTAP asignando un ciclo de reloj por instrucción ensambladora, no se consideraron efectos de *pipeline*. En la práctica, se puede asumir que el CPI es de 1.3, más realista que el valor 1 configurado. Como resultado, se puede asumir un factor de corrección de k=1.3.

Como se introdujo en la sección 3.2.1, el procesador *mMIPS* tiene un conjunto reducido de instrucciones, lo que significa que algunas operaciones se realizan por *software*, como las operaciones de multiplicación, división, modulo, y punto flotante. Al emplear el compilador *GCC* para el *mMIPS*, éste no es consciente de esas operaciones realizadas por *software*, por lo que introduce un error considerable. Por tanto, las

instrucciones ensambladoras proporcionadas por el compilador no se ajustan al número real que el compilador *LCC* emplea. Como resultado, la herramienta CTAP anota un número de instrucciones ensambladoras mucho menor, introduciendo un error.

8.5.5 Corrigiendo el error introducido por el compilador

Debido al problema que la herramienta CTAP tiene con el compilador *LCC* C del *mMIPS*, se corrigió el error utilizando en su lugar el compilador *GCC*. La solución propuesta para solventar el problema con las funciones realizadas en *software* fue la de añadir manualmente los ciclos de reloj para esas declaraciones. Para abordar ese asunto fue necesario medirlas mediante la instrumentación de la especificación RTL. El número de líneas de código que emplean dichas funciones es bajo, sin embargo, son comúnmente utilizadas en bucles. Por tanto, esto facilita su identificaron y medición. Obsérvese que esta circunstancia sólo ocurre en la aplicación JPEG y no en la librería *stdcomm*, donde las declaraciones fueron previamente sustituidas por otras implementadas en *hardware*, tal y como se describe en la sección 7.3.

En primer lugar, se aplicó la herramienta CTAP en los ficheros de la aplicación JPEG y en la librería *comm*. Seguidamente, se identificaron las declaraciones donde se encontraban las operaciones realizadas por *software*. En cada declaración identificada se añadieron *time stamps* antes y después de la instrucción *TIME_STAMP*. Posteriormente, la aplicación JPEG se ejecutó en el *Mini-NoC* y la información temporal se mostró por la salida estándar. El número total de ciclos de reloj se obtuvo a partir de la resta de ambos instantes de tiempo. Se realizó una media con los resultados obtenidos cada vez que dicha declaración era ejecutada. Finalmente, la media fue añadida al modelo anotado con CTAP mediante la sustitución de la estimación proporcionada por CTAP para esa declaración, tal y como se indica en la línea 6 de la *Figura 8.18*. Este proceso se repitió para un total de 67 declaraciones.

```
1  //STEP 2:       IDCT
2  …
3  #define CMUL(C, x)  (((C) * (x) + (1 << (C_BITS-1))) >> C_BITS)
4  DELAY_CYCLES(693);
5  z1[2] = SUB(CMUL( 8867, Y[2]), CMUL(21407, Y[6]));
6  DELAY_CYCLES(676); // add manually
7  z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
8  DELAY_CYCLES(k*3); // provided by CTAP
9  z3[1] = z2[1];
10 DELAY_CYCLES(k*3);
11 z3[2] = z2[2];
....
```

Figura 8.18: Ejemplo de la anotación manual empleada en la corrección del error del compilador

La ventaja de esta anotación es que el modelo anotado es independiente de la imagen de entrada. Este hecho permite evaluar ampliamente las prestaciones del modelo siendo posible hacer una comparativa con distintas imágenes.

8.5.6 Resultados

En la *Tabla 8.1* se presenta el tiempo de simulación empleado por la especificación RTL y los tres modelos anotados para decodificar la imagen *surfer.jpg*. El método utilizado para medir el tiempo simulación se explica en detalle en la sección 9.3.1.1.

RTL	***CASSE Mixed-Level* model**		
	***mMIPS* Instrumentation**	**CTAP**	**CTAP with the compiler error correction**
6 086 649 cc	6 432 487 cc	2 036 313 cc	6 011 116 cc

Tabla 8.1: Precisión de los distintos modelos anotados.

De acuerdo con los resultados presentados en la *Tabla 8.1*, el modelo anotado que corrige el error introducido por el compilador es el modelo más preciso (98.7%), ya que realiza una estimación de cada línea de código, mientras que la aproximación en la que se instrumenta el *mMIPS* pierde precisión, ya que la estimación se hace para largas partes de código. Aun así, el resultado se acerca al tiempo de simulación necesario en

el modelo a nivel RTL aunque no es tan próximo como el modelo con la anotación corregida.

Al final, se decidió utilizar la opción que emplea CTAP (con la corrección del error introducido por el compilador), ya que proporciona un modelo con una anotación que permite una evaluación de diferentes imágenes, mientras que la opción en la que se instrumenta el *mMIPS* no es posible, ya que depende de los datos de los que la información temporal ha sido obtenida. Además, resulta ser la opción más probable dentro de un contexto de diseño real, y la más interesante desde un punto de vista de diseño y herramienta.

8.6 Conclusiones

La primera fase de la implementación del modelo *Mixed-Level* consistió en modelar la aplicación en la herramienta *CASSE*. Esto requirió adaptar las primitivas de comunicación con el fin de cumplir con el protocolo de comunicación de *CASSE*. Para ello se dedicó un esfuerzo considerable, si bien, una vez adaptadas permiten la comunicación entre nodos del modelo y pueden ser utilizadas en cualquier aplicación.

La arquitectura comenzó a modelarse en los nodos procesadores a partir de componentes genéricos que la herramienta proporciona y que pueden ser instanciados e interconectados con metodología *plug and play*. El siguiente paso fue la adaptación de la red *mNoC* al modelo, lo cual requirió desarrollar un módulo adaptador que cambiara los niveles de abstracción haciendo posible la compatibilidad del protocolo ICCP de *CASSE* con los módulos descritos a nivel RTL.

Una vez modeladas las primitivas y la arquitectura, el tercer paso fue mapear la aplicación JPEG. Como *CASSE* permite el mapeo directo de la aplicación a la arquitectura, este paso se realizó de forma rápida y sencilla.

La fase final de la implementación del modelo correspondió a la anotación del tiempo en la aplicación. Para ello fueron exploradas varias opciones y, finalmente, se eligió emplear la herramienta CTAP para obtener la información temporal. El modelo

resultante proporciona una anotación independiente de los datos, lo que permite analizar las prestaciones del modelo mediante la comparativa con distintas imágenes, permitiendo una evaluación completa. A parte de la información proporcionada por CTAP, fue necesario realizar algunas anotaciones manuales en partes específicas de la aplicación. Esta aportación fue necesaria al no ser posible utilizar el compilador del *mMIPS* y en su lugar se empleó el compilador *GCC*, que no proporcionaba una correcta estimación temporal para ciertas instrucciones ensambladoras. A pesar de este inconveniente, el problema logró solucionarse satisfactoriamente.

Capítulo 9

Comparativa

9.1 Introducción

Una vez descrita la plataforma *Mini-NoC* y el modelo *CASSE Mixed-Level*, desarrollado en este Proyecto Fin Carrera es el momento de presentar la comparativa realizada entre ambas plataformas.

Este capítulo comienza definiendo las métricas identificadas para la realización de la comparativa. En segundo lugar, se presentarán las medidas realizadas en ambas plataformas y la comparativa entre ellas. El capítulo finaliza con las conclusiones obtenidas a partir del análisis de los resultados.

9.2 Definición de las métricas

Esta sección presenta las métricas propuestas para la comparativa de ambas plataformas, sus diferencias, y las aplicaciones propuestas para su obtención.

Las métricas definidas para la comparativa de la simulación del *Mini-NoC* (especificación RTL) y el modelo *CASSE* son:

- Prestaciones del simulador SystemC para el modelo CASSE *Mixed-Level* y la plataforma *Mini-NoC*. Es decir, el número de ciclos de reloj que son simulados por segundo.

- Latencia en el procesamiento de cada macrobloque que conforma la imagen. Es decir, el número de ciclos de reloj transcurridos entre el comienzo y fin del procesado de cada macrobloque.

- Latencia de las primitivas *sc_send()* y *sc_receive()* para mensajes de distintos tamaños. Es decir, el número de ciclos de reloj que tardan dichas funciones.

Obsérvense las diferencias entre la primera y segunda métrica. La primera está dirigida al análisis de la precisión del tiempo de simulación total y de la velocidad de simulación del modelo implementado. La segunda métrica está orientada a analizar el tiempo de procesamiento de cada macrobloque de la imagen, por lo que proporcionará información temporal acerca de puntos intermedios de la simulación. Esta información permitirá realizar un análisis de la precisión en varias partes de la simulación, y además, nos permitirá conocer cuándo los datos son procesados. Por tanto, estas dos métricas ofrecerán información para evaluar la velocidad del modelo implementado, y además, aportarán información para analizar la precisión global e intermedia de éste.

La tercera métrica permite validar la arquitectura (nodos procesadores, adaptadores, *NOC*), y analizar las primitivas de comunicación *sc_send()* y *sc_receive()*. Esta métrica no sólo incluye la latencia de computación, sino que también incorpora el tiempo requerido en la comunicación entre nodos.

El decodificador JPEG es la aplicación seleccionada para realizar la primera y la segunda métrica. En cambio, las medidas tomadas en la última métrica se llevarán a cabo a través de una sencilla aplicación que envía y recibe mensajes entre dos nodos procesadores.

9.3 Evaluación de la métricas

Esta sección presenta los resultados obtenidos en la plataforma *Mini-NoC* y en el modelo *CASSE Mixed-Level* para cada una de las tres métricas definidas. Cada subsección indica el método seguido para la obtención de las medidas en cada plataforma, los resultados, y un análisis del modelo *CASSE Mixed-Level*.

9.3.1 Prestaciones del decodificador JPEG

Esta métrica mide las prestaciones de simulación del decodificador JPEG para la plataforma *Mini-NoC* y el modelo *timed* de *CASSE Mixed-Level*. En esta métrica se analizará la precisión global del modelo *Mixed-Level*, a partir del número de ciclos de reloj. En segundo lugar, se analiza el aumento de velocidad del modelo *CASSE* frente a la especificación RTL, midiéndose número ciclos de reloj por segundo empleados en cada plataforma para ejecutar la aplicación.

Para realizar una evaluación completa, las medidas se han realizado con siete imágenes de distintos tamaños, lo que proporciona suficiente información para analizar el comportamiento del modelo.

9.3.1.1 *Mini-NoC* (RTL)

Como se adelantó en la sección 3.4, la plataforma *Mini-NoC* permite obtener el tiempo total simulado de una aplicación en el sistema. Esta información es proporcionada mediante una función *time stamp* añadida al código, y se muestra al finalizar la ejecución de la aplicación a través de la salida estándar.

Por otro lado, el tiempo de simulación requerido por la aplicación no es proporcionado directamente por la plataforma, siendo esta información necesaria, ya que el objetivo es hallar el número de ciclos de reloj por segundo. Para abordar este asunto, se utilizó el comando *time* de Linux.

El comando *time* se añade a un programa específico, de forma que cuando la aplicación termina su ejecución, este comando escribe un mensaje en la salida estándar ofreciendo una serie de estadísticas temporales acerca del programa ejecutado. Estas estadísticas consisten en los tiempos de ejecución del programa (i) el tiempo real transcurrido entre la invocación y la terminación, (ii) el tiempo de CPU consumido por el usuario y (iii) el tiempo de CPU consumido por el sistema. En el nuestro caso, la estadística temporal más interesante es la dada por la información relacionada con el tiempo empleado por el usuario (ii) ya que ofrece sólo el tiempo de ejecución de la aplicación en el *host* (ver línea 8 de la *Figura 9.1*).

Una vez añadido este comando, el siguiente paso es ejecutar el decodificador JPEG en la plataforma *Mini-NoC*, siguiendo los pasos descritos en la sección 3.4. Cuando la aplicación finaliza su ejecución por la salida estándar se muestra: el tiempo real - empleado facilitado por el comando *time* (ver línea 11), y el tiempo total simulado - ofrecido por la salida de la función *sc_time_stamp()* de SystemC (ver línea 8).

```
1  //unnecessary lines omitted
2  …
3  dp_x1y0.netif data read 0 @ 58971630 ns
4  FINISHED x1y0 @ 58973610 ns
5  …
6  60860000/480000000 @ 60860010 ns PC: x0y0:0x28  x0y1:0x1ffc
   x1y0:0x28  x1y1:0x17375b4
7  FINISHED x0y1 @ 60868490 ns
8  ALL FINISHED @ 60868490 ns
9
10 real     48m56.034s
11 user     48m27.583s
12 sys      0m26.054s
```

Outcome of the *time* command

Figura 9.1: Extracto de la salida estándar obtenido en la decodificación de la imagen *surfer.jpg*

Esta métrica se ha realizado para distintas imágenes, sin embargo, como ejemplo, se presentan los resultados de la imagen *surfer.jpg* para mostrar los pasos seguidos para obteneción.

Con el tiempo simulado, y sabiendo que el período del reloj es de 10 ns, es posible obtener el número de ciclos de reloj que necesitó la aplicación para simularse.

Por ejemplo, el tiempo simulado para el caso de la imagen *surfer.jpg* fue de 60868490ns, por tanto, el número de ciclos de reloj fue:

$$\text{Number of clock cycles to simulate the application} = \frac{60868490}{10} = 6086849\ clock\ cycles$$

La decodificación de la imagen *surfer.jpg* en el *Mini-NoC* necesitó de **48m 27.583s** (2907.583s) en un procesador Pentium IV corriendo en un a GNU/Linux a 4096 MB of RAM (co5.ics.ele.tue.nl).

Finalmente, el número de ciclos de reloj por segundo fue obtenido mediante la división del número de ciclos de reloj, entre el tiempo de simulación empleado en decodificar la imagen:

$$\text{Number of clock cycles simulated per second of computer time} = \frac{6086849}{2907.583} = 2093{,}43\ \frac{clock\ cycles}{sc} \cong 2{,}1\ kHz$$

9.3.1.2 Modelo *CASSE Mixed-Level*

Una vez que la métrica ha sido obtenida en la especificación RTL, el siguiente paso es realizarla en el modelo de *CASSE*. Toda la información requerida para obtenerla se puede obtener directamente de la salida estándar que *CASSE* proporciona cuando la simulación termina. En la *Figura 9.2* se muestra un ejemplo de la salida del modelo *CASSE Mixed-Level* después de decodificar la imagen. Observe que los datos de las prestaciones de simulación se muestran al final de la simulación y requiere de la división por el reloj (10 ns) para obtener el número total de ciclos de reloj. Finalmente, la información del tiempo de simulación es directamente ofrecida por la herramienta.

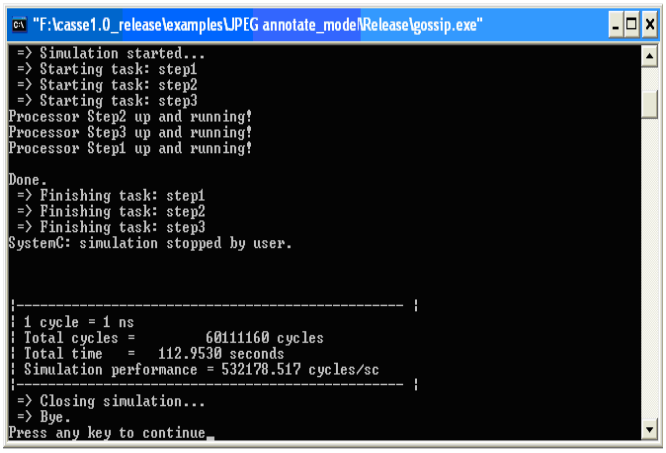


Figura 9.2: Salida de *CASSE* después de la decodificación de la imagen *surfer.jpg*

9.3.1.3 Resultados y Evaluación

En la *Tabla 9.1* se presentan las medidas realizadas en la plataforma *Mini-NoC* y en el modelo *CASSE* sobre las imágenes consideradas. La primera columna de la tabla lista las imágenes de entrada, ordenadas en función de su tamaño de mayor a menor. La segunda columna enumera los diferentes parámetros de la métrica, como el número total de ciclos de reloj, el tiempo de simulación, y el número de ciclos de reloj por segundo requerido en decodificar la imagen para cada modelo. Los dos campos siguientes representan la evaluación de la precisión global y el aumento de velocidad del modelo *CASSE Mixed-Level* frente a la especificación RTL.

Input Image	**Parameters**	***Mini-NoC* (RTL)**	***CASSE Mixed-Level***
Gpyramid.jpg (80x60)	Clk cycles	22 097 001	20 878 720
	Simulation Time	319 m	390 sc
	Simulation Performance	1.1k	53.5k
	Accuracy	100%	94.4%
	RTL speedup (x times)	1	48.6

Tulip.jpg (38x48)	Clk cycles	14 137 610	13 302 198
	Simulation Time	116 m	249 sc
	Simulation Performance	2k	53.4k
	Accuracy	100%	94.1%
	RTL speedup (x times)	1	26.7
Pyramid.jpg (42x22)	Clk cycles	9 293 648	8 827 169
	Simulation Time	96 m	164 sc
	Simulation Performance	1.6k	53.5k
	Accuracy	100%	95%
	RTL speedup (x times)	1	33.4
Surfer2.jpg (30x32)	Clk cycles	7 184 050	6 592 221
	Simulation Time	77 m	129 sc
	Simulation Performance	1.5k	50k
	Accuracy	100%	91.7%
	RTL speedup (x times)	1	33.3
Surfer (32x24)	Clk cycles	6 086 649	6 011 116
	Simulation Time	48 m	113 sc
	Simulation Performance	2k	53.2k
	Accuracy	100%	98.7%
	RTL speedup (x times)	1	26.6
Pc.jpg (16x16)	Clk cycles	2 379 982	2 341 000
	Simulation Time	26 m	43 sc
	Simulation Performance	1.5k	54.1k
	Accuracy	100%	98.3%
	RTL speedup (x times)	1	36

Zw.jpg (8x8)	Clk cycles	902 963	894 544
	Simulation Time	9 m	17 sc
	Simulation Performance	1.6k	52.9k
	Accuracy	100%	99%
	RTL speedup (x times)	1	33

Tabla 9.1: Comparativa de las prestaciones de ambos modelos con imágenes de distintos tamaños

Los resultados de la imagen de mayor tamaño, *Gpyranid.jpg*, muestran que la precisión del modelo es del 94.4%, y la velocidad de simulación en el modelo *Mixed-Level*, es 53.5 Kciclos/seg frente a 1.1 Kciclos/seg en RTL. Por otro lado, la especificación RTL tardó 319 minutos, mientras el modelo *Mixed-Level* solamente necesitó de 6 minutos. Por tanto, a pesar de que el modelo *CASSE* soporta una red descrita en RTL, la velocidad de simulación aumentó en 48.6 Kciclos/seg, y mejoró en 313 minutos el tiempo de simulación.

Los datos de la *Tabla 9.1* revelan que, a medida que el tamaño de la imagen aumenta el número de ciclos de reloj empleados en su decodificación también se incrementa. Esta es la razón por la que se han incluido imágenes de gran tamaño, ya que al requerir mayor computación pueden repercutir en una disminución de la precisión del modelo. Como se muestra en la *Tabla 9.1*, esta circunstancia no sucede en el modelo de *CASSE Mixed-Level*, donde la precisión ronda el 95% y permanece constante en las imágenes de mayor tamaño.

Esta tendencia se evidencia en la *Figura 9.3*, donde se muestra el número total de ciclos de reloj requerido para cada una de las siete imágenes decodificadas en ambos modelos. Observe que el error permanece constante a lo largo de las diferentes imágenes a pesar de que algunas requieren mayor capacidad de cómputo.

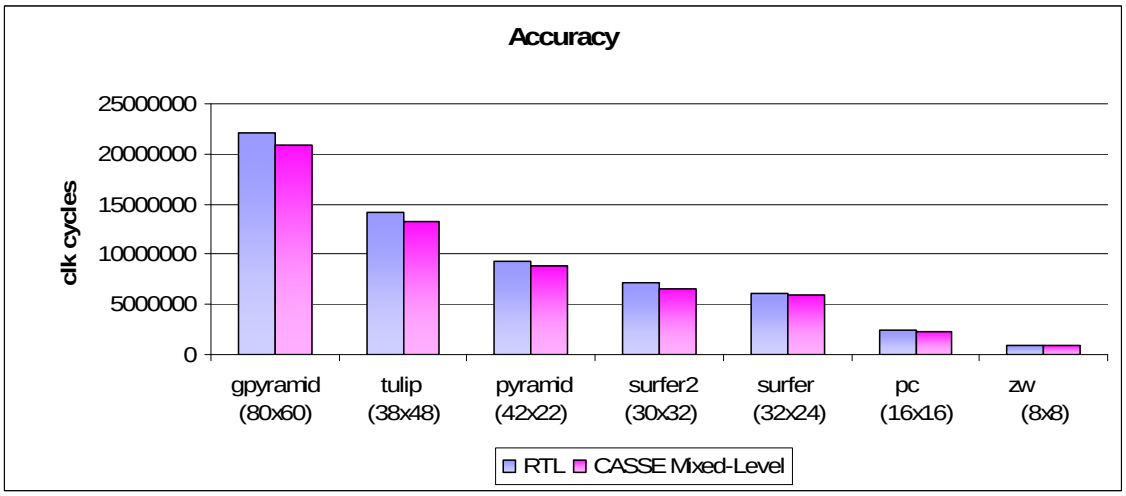


Figura 9.3: Evaluación de la precisión

La *Figura 9.4* presenta el tiempo de simulación (en segundos) necesario para la decodificación de las siete imágenes. Como se ilustra en escala logarítmica, el tiempo de simulación para el modelo *Mixed-Level* es, al menos, un orden de magnitud más rápido que en la simulación RTL.

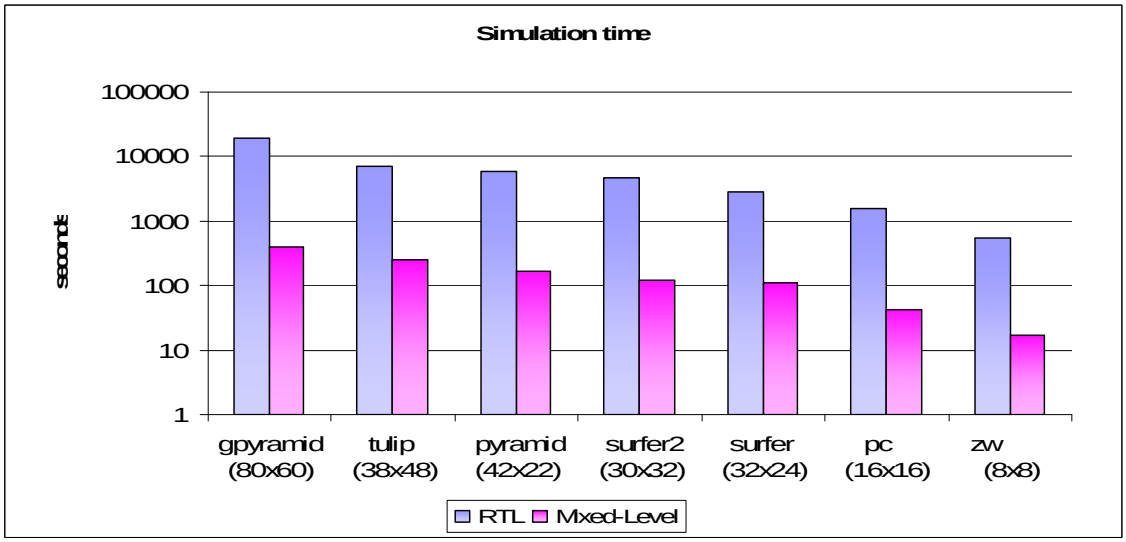


Figura 9.4: Comparativa del tiempo de simulación

A partir de los resultados experimentales se deduce que la velocidad de simulación se incrementa con un factor medio de 34 comparado con la ejecución RTL.

La *Figura 9.5* muestra en escala logarítmica las prestaciones de la simulación de las siete imágenes. Observe que a la imagen de mayor tamaño le corresponde una velocidad de simulación más baja de la especificación RTL, en esta caso 1.1kcc/seg.

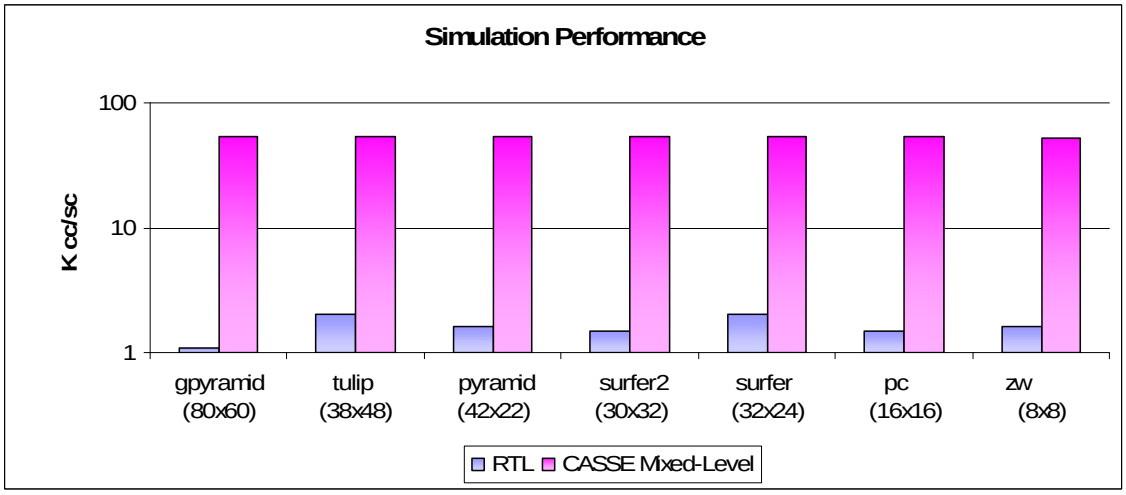


Figura 9.5: Comparativa de la velocidad de simulación

Para concluir, el modelo *Mixed-Level* es 34 veces más rápido que el modelo RTL manteniendo una precisión media del 96%.

9.3.2 Latencia en el procesado de cada macrobloque

Como se introdujo en la sección 6.2, el proceso de decodificación ha sido dividido en tres tareas, cada una de las cuales se ha mapeado en uno de los nodos *mMIPS* del *Mini-NoC*. El decodificador toma la imagen y la divide en macrobloques, que a su vez se dividen en bloques. La división en bloques se realiza en el nodo 1, siendo estos procesados y enviados a través del *mNoC*. Los bloques son recibidos y procesados en el nodo 2, y seguidamente reenviados al nodo 3, donde también son procesados y reordenados en macrobloques.

El objetivo de esta métrica es hallar el número de ciclos de reloj entre el comienzo de procesado del primer macrobloque y su finalización, para todos los macrobloques. Es decir, el número de ciclos de reloj transcurridos desde que el nodo 1 procesa el macrobloque hasta que éste es recibido y procesado en el nodo 3. Como los

bloques necesitan ser transferidos a través del nodo 2, la métrica implica no sólo tiempo de computación, sino también de comunicación.

Además, también se realizará una comparativa acerca del momento en el que cada macrobloque es procesado. Mientras la latencia proporciona información para realizar un análisis de la precisión en momentos intermedios de la aplicación, a partir la información temporal es posible evaluar cuándo son procesados los datos. Estos dos parámetros proporcionaran suficiente información sobre el comportamiento del modelo en instantes intermedios de la simulación.

Como en cada sección, primeramente se describirán el procedimiento seguido en para realizar las medidas en ambos modelos, a continuación los resultados y por último el análisis de estos. Las medidas se han realizado para un total de tres imágenes de distintos tamaños.

En el presente Resumen sólo se muestran los resultados obtenidos para una de las imágenes, mientras que en las secciones 8.3.2.3.2 y 8.3.2.3.3 de la Memoria se presentan los resultados de las dos restantes.

9.3.2.1 *Mini-NoC* (RTL)

Para obtener la latencia, se ha empleado la instrucción *TIME_STAMP*. Como se explicó en la sección *7.2*, esta instrucción fue implementada para obtener el tiempo actual de la aplicación en una línea específica de código.

El primer paso fue añadir dos *time stamps*, uno en el nodo 1 y otro en el nodo 3, es decir, cuando la aplicación comienza y finaliza el procesamiento del macrobloque. Seguidamente, se ejecutó el decodificador JPEG en la plataforma *Mini-NoC*. La información temporal se mostró en la salida estándar durante la ejecución y la latencia se obtuvo mediante la substracción de ambos *time stamps.* Este proceso se ha realizado en todos los macrobloques que conforman la imagen. Como ejemplo, se muestra en la *Figura 9.6* cómo esta latencia fue medida para la imagen *surfer.jpg*.

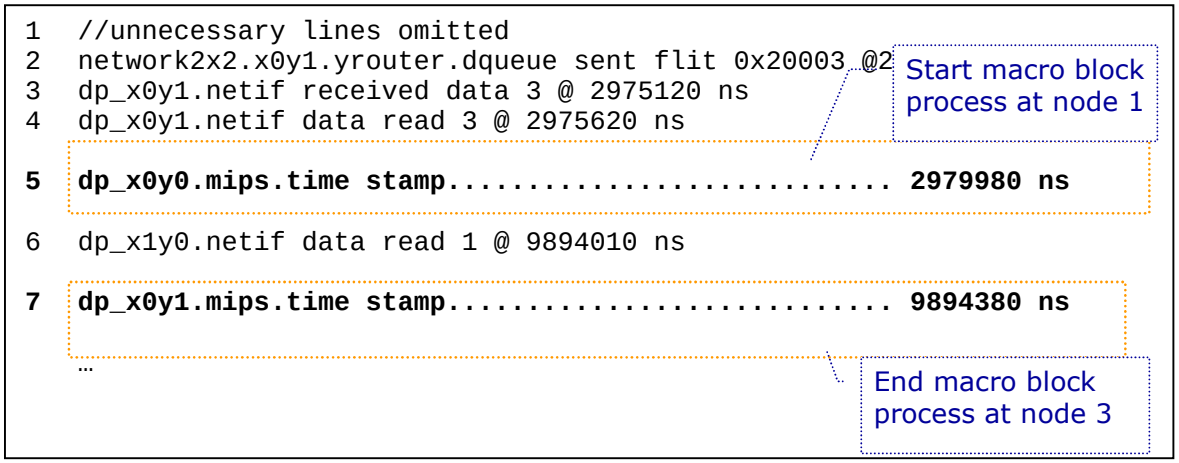


Figura 9.6: Extracto de la salida estándar

9.3.2.2 Modelo *CASSE Mixed-Level*

En el modelo *CASSE Mixed-Level* la latencia también fue obtenida añadiendo *time stamps* en los nodos 1 y 3 mediante la función *sc_time_stamp()* de SystemC. Cuando la aplicación se ejecuta, la información temporal se muestra en la salida estándar de *CASSE*, y la latencia es obtenida mediante la substracción de ambos *time stamps*.

9.3.2.3 Resultados de la latencia de cada macrobloque

Los experimentos se han realizado en un total de tres imágenes. Como sus tamaños son diferentes, el número de macrobloques requeridos en la decodificación de la imagen también varía entre ellas. Esto permite realizar una comparativa con un número de macrobloques que varía entre 12 y 4, en los que los puntos de comienzo y final de procesado ocurren en un amplio rango. Los resultados aportan información de cómo el modelo se comporta para distintas imágenes.

Como se explicó anteriormente, es posible obtener el tiempo de procesado de cada macrobloque mediante la sustracción entre el inicio y el fin del procesado de cada uno. Una vez obtenido, se halla el número de ciclos de reloj dividiendo el resultado previo por el período del reloj (10ns). Este proceso se siguió en todos los macrobloques que componen las tres imágenes en ambos modelos

9.3.2.3.1 Imagen *Surfer.jpg*

Los resultados obtenidos en la decodificación de la imagen *surfer.jpg* en la especificación RTL y en el modelo *Mixed-Level*, se indican en las *Tablas 9.2* y *9.3*. La primera columna enumera cada macrobloque que conforma la imagen. La segunda y tercera columna listan el tiempo inicial y final de procesamiento. La última columna contiene la latencia, es decir, el número total de ciclos de reloj empleados en el procesamiento de cada macrobloque.

Macro block	Start processing	End processing	Clock cycles
0	2 979 980 ns	9 894 380 ns	**691 440**
1	6 815 020 ns	14 512 450 ns	**769 743**
2	11 434 210 ns	19 130 920 ns	**769 671**
3	16 051 930 ns	23 750 820 ns	**769 892**
4	20 671 220 ns	28 370 820 ns	**769 960**
5	25 291 240 ns	33 146 070 ns	**785 483**
6	30 072 350 ns	37 765 290 ns	**769 294**
7	34 686 000 ns	42 385 690 ns	**769 969**
8	39 305 970 ns	47 006 020 ns	**770 232**
9	43 925 450 ns	51 625 630 ns	**770 018**
10	48 545 750 ns	56 245 400 ns	**769 965**
11	53 165 000 ns	60 866 140 ns	**770 114**

Tabla 9.2: Latencia de cada macrobloque en RTL para la imagen *surfer.jpg*

Los resultados revelan que cada macrobloque necesita cerca de 700k ciclos de reloj para procesarse. Observe que la latencia del primer macrobloque es la menor, siendo esto debido a que el *pipeline* de la red está vacío, por lo que la latencia del resto de macrobloques incluye también el tiempo transcurrido en *buffers* intermedios donde se espera a que el macrobloque anterior termine su procesamiento en los nodos.

Macro block	Start processing	End processing	Clock cycles
0	2 990 930 ns	9 784 250 ns	**679 332**
1	6 953 840 ns	14 305 230 ns	**735 139**
2	11 474 820 ns	18 826 160 ns	**735 134**
3	15 995 740 ns	23 347 190 ns	**735 145**
4	20 516 760 ns	27 868 190 ns	**735 143**
5	25 037 650 ns	32 713 770 ns	**767 612**
6	29 883 320 ns	37 318 400 ns	**743 508**
7	34 487 930 ns	41 839 230 ns	**735 130**
8	39 008 840 ns	46 360 330 ns	**735 149**
9	43 529 780 ns	51 069 210 ns	**753 943**
10	48 238 740 ns	55 590 140 ns	**735 140**
11	52 759 740 ns	60 111 070 ns	**735 133**

Tabla 9.3: Latencia de cada macrobloque en el modelo *CASSE Mixed-Level* para la imagen *surfer.jpg*

Para permitir un análisis comparativo de las *Tablas 9.2* y *9.3*, en la *Figura 9.7* se representa el comienzo y final de cada macrobloque en ambas plataformas. Observe que el modelo implementado es bastante preciso, ya que comienza y finaliza en el mismo orden de magnitud.

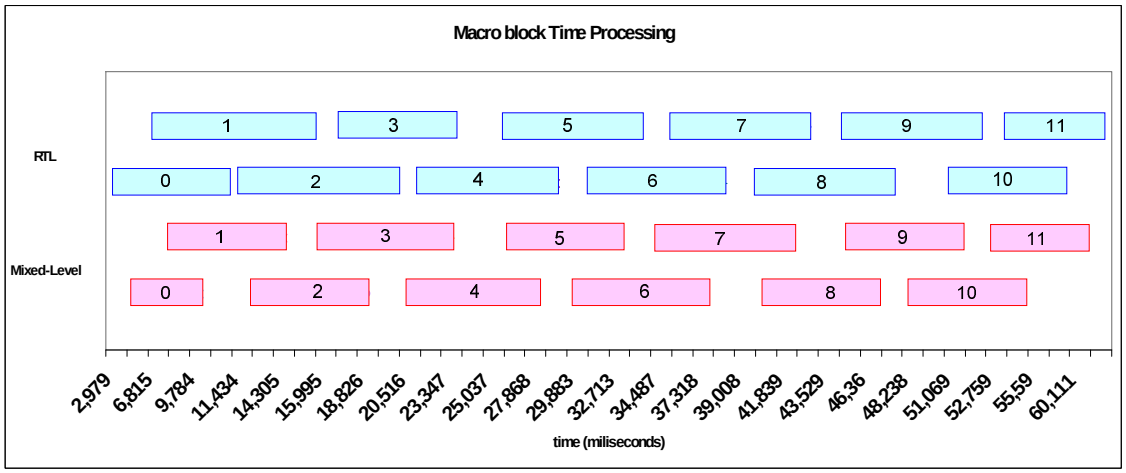


Figura 9.7: Comparativa del tiempo de procesamiento para la imagen *surfer.jpg*

Observe que el punto de comienzo del procesamiento del primer macrobloque en el modelo *Mixed-Level* se acerca al RTL (99,6% precisión). Este hecho es importante, ya que antes de que el primer macrobloque sea procesado, gran parte del proceso de

computación se realiza en el nodo 1. Es más, la computación del nodo 1 tiene gran dependencia de los datos, es decir, que dependiendo de la imagen de entrada se requiere más o menos tiempo de computación. Por tanto, el error introducido en este punto es útil ya que permite estimar la perdida de precisión del modelo. Observe que si se corrige dicho error, cada macrobloque de la imagen comenzaría casi al mismo momento que lo hace en la especificación RTL.

Con la información de las latencias de las tablas anteriores, se ha realizado una evaluación de la precisión que se muestra en la *Tabla 9.4*. La primera fila enumera los macrobloques que conforman la imagen. La segunda y tercera columna representan el número de ciclos de reloj requeridos para procesar cada macrobloque en ambos modelos. La siguiente columna muestra la diferencia de ciclos de reloj en ambos modelos. Finalmente, la última columna muestra la precisión del modelo *Mixed-Model* frente a la especificación RTL para cada macrobloque.

Macro block	RTL	Mixed-Level	Clk Cycle Diff	Accuracy (%)
0	691 440	679 332	12 108	98,4
1	769 743	735 139	34 604	95,5
2	769 671	735 134	34 537	95,5
3	769 892	735 145	34 747	95,4
4	769 960	735 143	34 817	95,4
5	785 483	767 612	17 871	97,7
6	769 294	743 508	25 786	96,4
7	769 969	735 130	34 839	95,4
8	770 005	735 149	34 856	95,4
9	770 018	753 943	16 075	97,9
10	769 965	735 140	34 825	95,4
11	770 114	735 133	34 981	95,4

Tabla 9.4: Comparativa de latencias de los macrobloques para la imagen *surfer.jpg*

Observando la *Tabla 9.4* se deduce que el número de ciclos de reloj difiere de 12k a 34k. Como resultado, el modelo implementado alcanza una precisión del 96,1% de media.

La *Figura 9.8* representa la latencia de cada macrobloque para ambos modelos. Observe que el modelo *Mixed-Level* sigue la misma tendencia que el modelo RTL, incluyendo además incluye el efecto de *pipeline* de la red que sucede también en el primer macrobloque del modelo RTL.

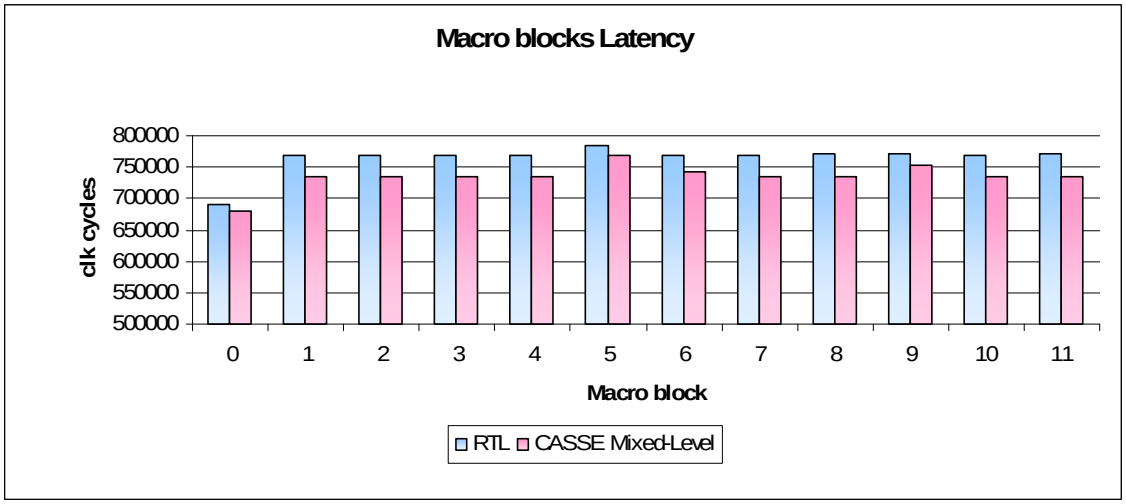


Figura 9.8: Comparativa de las latencias de los macrobloques para la imagen *surfer.jpg*

9.3.3 Latencia en el envío y recepción de un mensaje

El objetivo de esta métrica es la de conocer el número total de ciclos de reloj entre el comienzo del envío del mensaje y la completa de recepción de éste.

9.3.3.1 *Mini-NoC* (RTL)

El primer paso fue codificar una simple aplicación y ejecutarla en el *Mini-NoC*. La aplicación, denominada *Test*, envía un mensaje desde el nodo x0y0 al nodo x1y0, tal y como se representa en la *Figura 9.9*Por tanto, sólo se ejecuta una función *sc_send()* en el nodo (0,0), y un *sc_receive()* en el nodo (1,0) (ver líneas 9 y 13 de la *Figura 9.10*). El mensaje se divide en palabras de 4 bytes que son enviadas a la interfaz de red, y reenviadas posteriormente a la red. La función *sc_receive()* recolecta todas la palabras y reconstruye el mensaje. Como los datos sólo provienen del nodo (0,0) no se produce contención en la red.

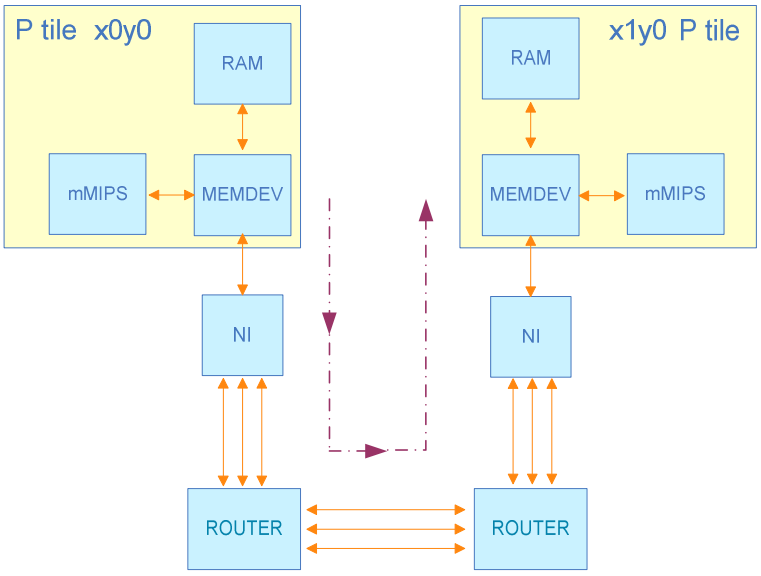


Figura 9.9: Módulos de la plataforma *Mini-NoC* empleados en la aplicación

Para hallar el número de ciclos de reloj requeridos por las funciones *sc_send()* y *sc_receive()*, se empleó en la instrucción *TIME_STAMP*. La medida se realizó añadiendo esta instrucción antes y después de la función. Una vez que la simulación finalizó, la latencia fue obtenida sustrayendo el valor de ambos *time stamps*. El uso de esta instrucción permite añadir precisión a la medida, ya que el interés es hallar la latencia de las funciones, y no de la aplicación.

```
1   #include "stdcomm.h"
2   //unnecessary lines omitted
3   …
4   void main (void) {
5     my_node_number =  *((int *)0x00);
6     rela_dest_addr = *((int *)0x04);
7
8     if (my_node_number == 0x00) {
9       sc_send(rela_dest_addr, send_str_1, 4);
10
11    else {
12      if (my_node_number == 0x01){
13         sc_receive(received_str_1, 4);
14      } };
```

Node (0,0) sends the message

Node (1,0) receives the message

Figura 9.10: Ejemplo de la aplicación *Test* para el envío de un mensaje con una palabra

9.3.3.2 Modelo *CASSE Mixed-Level*

Al igual que en el *Mini-NoC*, se empleó una sencilla aplicación que envía un mensaje desde el nodo x0y0 al nodo x1y0, tal y como se ilustra en la *Figura 9.11*. Por tanto, sólo es necesario ejecutar una función *sc_send()* en el nodo (0,0), y una función *sc_receive()* en el nodo (1,0). Las palabras en la red provienen sólo del nodo x0y0, por lo que no se produce contención en la red.

La medida se ha realizado añadiendo *time stamps* al final de las funciones *sc_send()* y *sc_receive()* de la aplicación. Una vez que la simulación finalizó, en la salida estándar se obtuvo la latencia.

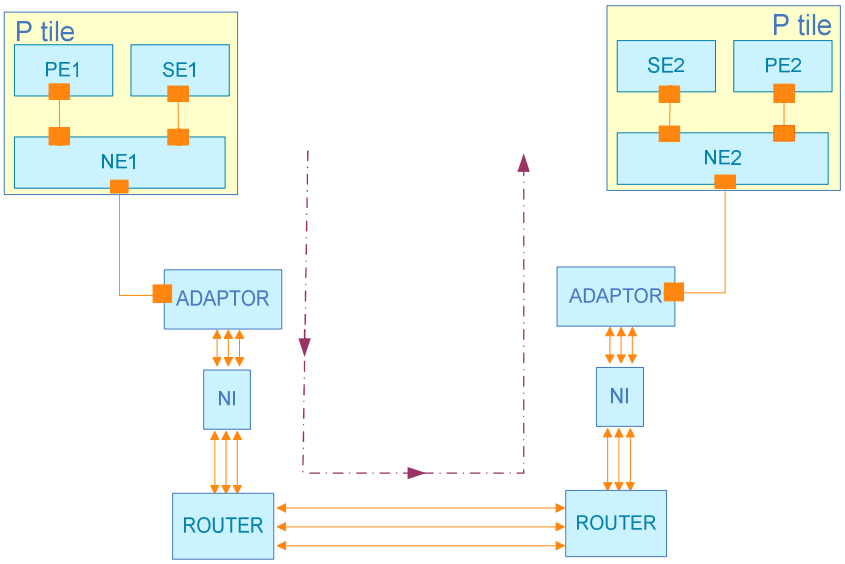


Figura 9.11: Módulos del modelo *CASSE Mixed-Level* usados en la aplicación

9.3.3.3 Resultados de latencia en el envío de un mensaje

Las medidas se han realizado para mensajes de 1 palabra (4 bytes), 2 (8 bytes), 4 (16 bytes), 8 (32 bytes), 16 (64 bytes), 32 (128 bytes), y 64 (256 bytes). La *Tabla 9.5* indica las latencias de la primitiva *sc_send()* medidas en la plataforma *Mini-NoC* y en el modelo *CASSE Mixed-Level*.

La primera fila representa el número de palabras enviadas en mensajes de distinto tamaño. Las siguientes listan el número de ciclos de reloj que la función *sc_send()* empleó en ambas plataformas.

Primitive	Models	1	2	4	8	16	32	64
sc_send	**RTL**	311	571	1091	2131	4211	8371	16691
	Mixed-Level	317	597	1157	2277	4517	8997	17957

Tabla 9.5: Latencias de la primitiva *sc_send()* para mensajes de distintos tamaños en ambas plataformas

A partir de la *Tabla 9.5* se deduce que el error introducido aumenta a medida que el número de palabras enviadas en el mensaje aumenta. Realizando un análisis del error relativo introducido en los mensajes, la conclusión alcanzada es que, desde una perspectiva global, el error aumenta ligeramente, como se representa en la *Tabla 9.6*. Este hecho es debido a que el tamaño del código ejecutado y por tanto el error relativo, aumenta a medida que se incrementa el número de palabras.

Packets	1	2	4	8	16	32	64
Relative Error	$1.9 \cdot 10^{-2}$	$6 \cdot 10^{-2}$	$6.8 \cdot 10^{-2}$	$7.2 \cdot 10^{-2}$	$4.5 \cdot 10^{-2}$	$7.4 \cdot 10^{-2}$	$7.5 \cdot 10^{-2}$

Tabla 9.6: Error relativo de la latencia de la primitiva *sc_send()* en el modelo *Mixed-Level*

Observe que en la implementación RTL, a medida que en el mensaje se dobla el número de palabras, la latencia también se duplica. Este hecho también sucede en el modelo *Mixed-Level*. Por tanto, la *Tabla 9.5* ilustra cómo las tendencias de ambas plataformas son similares. Esta característica puede observarse en la *Figura 9.12*, en la que se presentan las latencias en de la primitiva *sc_send()*ambas plataformas en escala logarítmica.

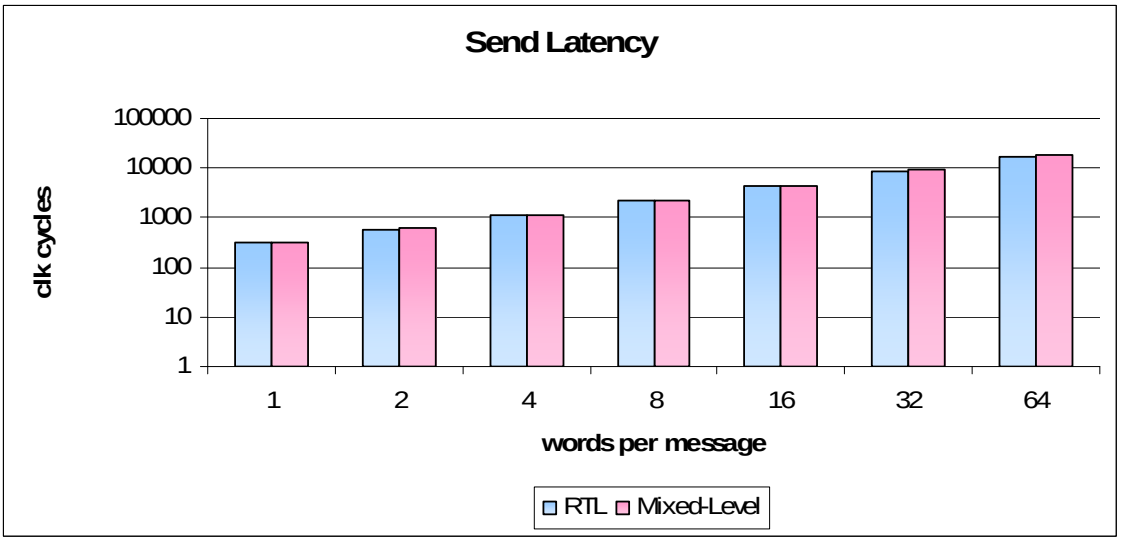


Figura 9.12: Comparativa de las latencias de la primitiva *sc_send()* en ambos modelos

El mismo análisis se ha realizado con la primitiva *sc_receive()*, cuyas latencias para los distintos mensajes se listan en la *Tabla 9.7*.

Primitive	Models	1	2	4	8	16	32	64
sc_receive	**RTL**	490	778	1260	2318	4387	8572	16895
	Mixed-Level	495	771	1334	2449	4690	9172	18136

Tabla 9.7: Latencias de la primitiva *sc_receive()* para mensajes de distintos tamaños en ambas plataformas

Observe que la función *sc_receive()* requiere más ciclos de reloj que la función *sc_send()*, ya que necesita reconstruir el mensaje y esto requiere de tiempo adicional. Esta característica también se puede observar en la latencia del modelo implementado. Sin embargo, como en la función *sc_send()*, el número de ciclos de reloj que se necesita la función *sc_receive()* se duplica a medida que el mensaje también lo hace. Esta tendencia se puede apreciar en la *Figura 9.13*. Además, en esta gráfica se puede observar la poca diferencia entre ambas latencias.

Packets	1	2	4	8	16	32	64
Relative Error	$1 \cdot 10^{-2}$	$0.8 \cdot 10^{-2}$	$5.8 \cdot 10^{-2}$	$5.6 \cdot 10^{-2}$	$6.9 \cdot 10^{-2}$	$6.9 \cdot 10^{-2}$	$7.3 \cdot 10^{-2}$

Tabla 9.8: Error relativo de la latencia de la primitiva *sc_receive()* en el modelo *Mixed-Level*

Realizando el mismo análisis del error relativo con la primitiva *sc_receive()*, se alcanza la misma conclusión: el error aumenta ligeramente a medida que se incrementa el tamaño del mensaje.

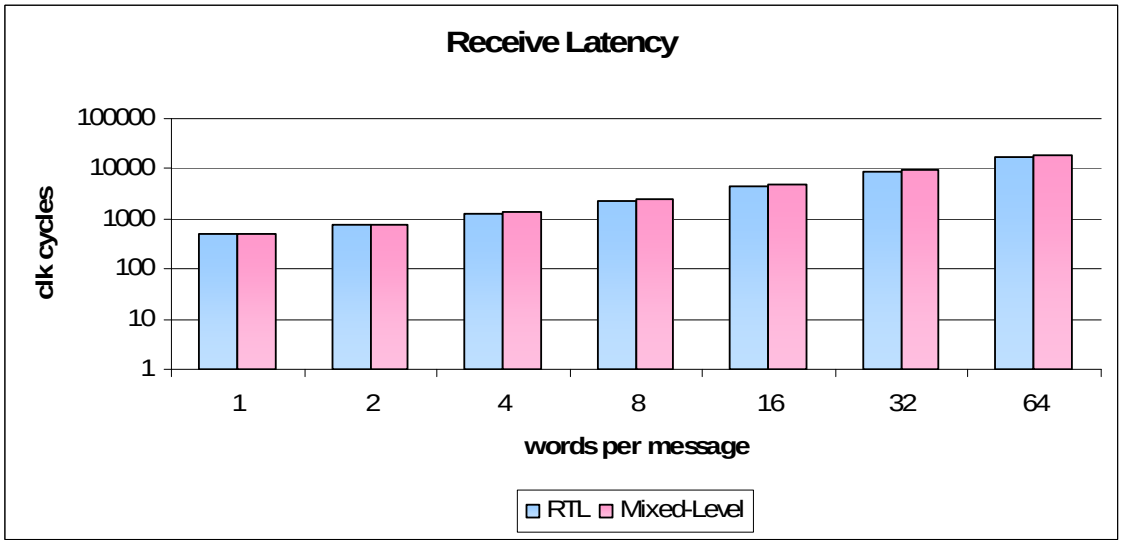


Figura 9.13: Comparativa de las latencias de la primitiva *sc_ receive()* en ambos modelos

9.4 Conclusiones

A partir de los resultados experimentales obtenidos en este análisis, se puede afirmar que el modelo de *CASSE Mixed-Level* es, en media, 34 veces más rápido que la especificación RTL, manteniendo una precisión del 96%.

El análisis de las características intermedias revela que la precisión media en diferentes partes de la aplicación es del 95.2%. Además, los datos son procesados en instantes muy próximos a los que son procesados en la especificación RTL. Otra conclusión extraída de este análisis es que la perdida de la precisión global es introducida en el nodo 1 debido a la alta dependencia de datos. Por tanto, dependiendo

de la imagen de entrada se requiere más o menos tiempo de computación, introduciendo así error en el modelo.

Finalmente, se puede concluir que la idoneidad del modelo ha sido validada, ya que representa fielmente las características del sistema. Las primitivas de comunicación del modelo *Mixed-Level* presentan la misma funcionalidad que las de la plataforma *Mini-NoC*, con una imprecisión inferior al 7%.

Capítulo 10

Conclusiones y Futuras Líneas de Trabajo

En este último capítulo de la memoria se hace un balance del trabajo realizado en el presente Proyecto Fin de Carrera, ofreciendo una serie de conclusiones y recomendaciones. Para concluir, se proponen líneas de trabajo futuras con las que continuar este Proyecto Fin de Carrera.

10.1 Conclusiones

El objetivo principal propuesto para este Proyecto Fin de Carrera, tal y como se describe en la sección 1.2, consistía en el desarrollo de un modelo mixto entre niveles de abstracción RTL y TLM que emulara la plataforma *Mini-NoC* usando el entorno *CASSE*. Dicho modelo mixto tenía que compararse con la referencia implementada en SystemC RTL en términos de velocidad de simulación, números de ciclos de ejecución y facilidad de modelado. Tal y como se describe a lo largo de este Proyecto Fin de Carrera dicho objetivo principal se ha conseguido.

El modelado de la plataforma *Mini-NoC* se realizó aprovechando las características del entorno *CASSE*, que permite describir el sistema como una composición modular de elementos predefinidos que son instanciados y configurados a través de un fichero de texto. Dicho fichero de texto permite una descripción simple y

natural de los elementos que componen la arquitectura y su configuración, sin necesidad de escribir manualmente ninguna línea de código en SystemC.

El mayor esfuerzo en el modelado de la plataforma se debió a la creación de adaptadores que permitieran la comunicación entre los nodos procesadores y la red *NoC*, ya que los nodos procesadores (TLM) y la red *mNoC* (RTL) estaban descritos en niveles de abstracción diferentes. Sin embargo, la mayor parte del esfuerzo empleado en este Proyecto Fin de Carrera no se invirtió en el modelado de la arquitectura, sino en la anotación de las tareas que componen la aplicación JPEG con la información temporal de ejecución. Esto fue debido principalmente a los problemas surgidos con el compilador *LCC* del procesador *mMIPS*, que impedía utilizar la herramienta CTAP. Aun así, dichos problemas fueron superados de forma satisfactoria utilizando el compilador *GCC* para el procesador *mMIPS* y reemplazando las instrucciones *software* del código JPEG y de la librería *stdcomm*.

De acuerdo con los resultados de la comparativa realizada, el modelo mixto *CASSE* en combinación con la herramienta CTAP, produjo unos resultados con una precisión de media del 96% en el tiempo de ejecución, y una mejora en la velocidad de simulación en un factor medio de 34. Es decir, para prácticamente el mismo número de ciclos de ejecución, las simulaciones con *CASSE* son 34 veces más rápidas, aun teniendo en cuenta que parte de la plataforma está todavía descrita a nivel RTL, lo cual ralentiza la simulación. A tenor de los resultados obtenidos se puede concluir que la herramienta CTAP es una combinación perfecta a la técnica de emulación de código utilizada en *CASSE*, y permitiendo reemplazar los simuladores de juego de instrucciones (ISS) sin un gran impacto en la simulación.

La conclusión más importante alcanzada en este trabajo de investigación es por tanto que la combinación del entorno *CASSE* y la herramienta CTAP permite una exploración sencilla y rápida del espacio de diseño en plataformas multiprocesadoras (*MP-SoC*) basadas en *NoC*. Este hecho se debe a la rapidez con la que se producen las simulaciones manteniendo un buen nivel de precisión en los resultados y eficiencia en el modelado. Además, este proyecto contribuye a demostrar que el tiempo dedicado a la validación funcional de plataformas complejas se puede reducir dramáticamente utilizando técnicas de abstracción mediante modelos SystemC TLM. Esta mejora está

presente incluso cuando se crean modelos mixtos TLM y RTL. Este hecho permite a los ingenieros verificar la implementación de bloques específicos de *hardware* a nivel RTL, mientras que el resto del sistema permanece a un nivel de abstracción mayor mediante modelos descritos en TLM.

10.2 Recomendaciones

Para una posible continuidad futura del presente trabajo de investigación, se proponen las siguientes cuestiones:

- Proporcionar funcionalidad adicional al compilador *LCC* del *mMIPS* para que soporte el formato estándar STABS de depurado. Esta modificación permitirá que el compilador pueda ser utilizado en CTAP.

- Extender el juego de instrucciones del procesador *mMIPS* con la implementación *hardware* de algunas instrucciones que actualmente se realizan por *software*. Esta extensión optimizaría las simulaciones de la especificación RTL.

10.3 Líneas de trabajo futuras

El trabajo de investigación realizado presenta posibilidades de futuro desarrollo. La línea de trabajo futura más interesante es la sustitución del componente *NOC* del modelo *Mixed-Level* por un nuevo componente *NOC* modelado a un nivel de abstracción mayor (TLM). De esta forma, el nuevo modelo de *CASSE* estaría descrito completamente en TLM, con lo que se conseguiría una importante aceleración en el tiempo de simulación.

La implementación propuesta para el nuevo componente *NOC* estaría compuesta por dos módulos básicos: *network interface* y *router*, que se representan en la *Figura 10.1*. Una vez modelados ambos módulos, estos elementos se conectarían siguiendo la arquitectura de la red *mNoC* de la plataforma *Mini-NoC*, mostrada en la *Figura 3.4*.

Para el módulo de la interfaz de red se propone que un hilo de ejecución controle las transacciones en ambos sentidos (entre nodos procesadores y *routers*). Además, se requiere de *buffers* emplazados en los puertos de entrada, cuya función es gestionar los datos de entrada desde la red, como se muestra en la *Figura 10.1*.

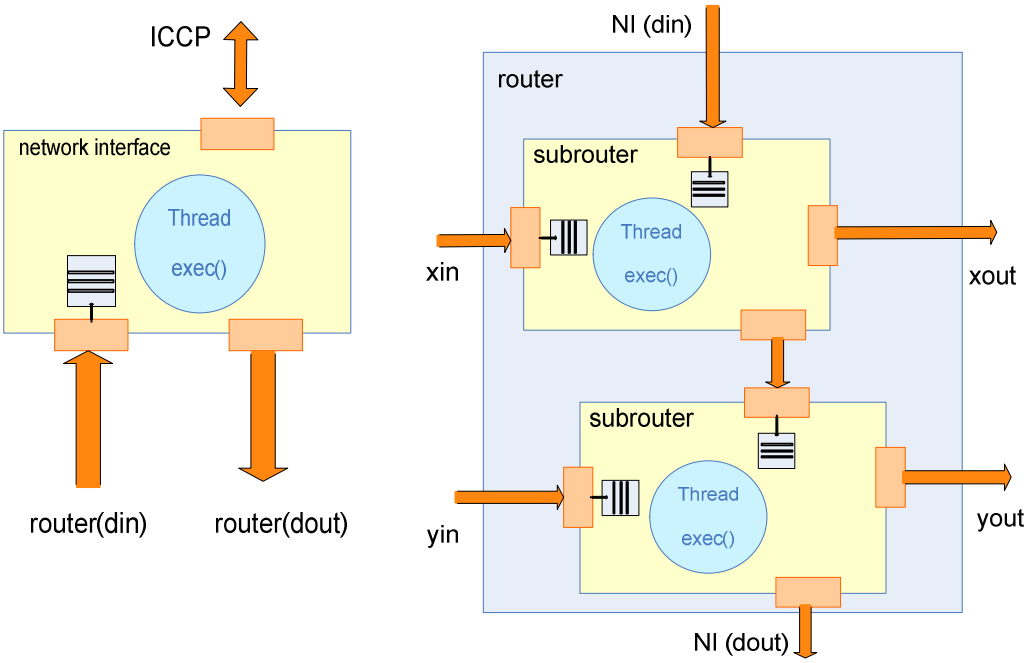


Figura 10.1: Módulos propuestos como línea futura de investigación

El *router* sería modelado basándose en la implementación original del *router* del *Mini-NoC*. Está compuesto por dos *subrouter* con idéntica funcionalidad. Cada uno está compuesto por un hilo de ejecución que gestiona los datos a través de *buffers*. La prioridad se establece mediante la aplicaron de un algoritmo de *round robin*, al igual que en el *router* original del *mNoC*.

Una característica importante del modelo del *router* es que cuando no hay datos para transferir en las colas de entrada, su hilo asociado se suspende y no se vuelve a reanudar hasta se dispone de un dato. Este hecho permite que la simulación se acelere al no consumir eventos en el núcleo del simulador de SystemC.

Presupuesto

El presupuesto de este Proyecto Fin de Carrera se desglosa en los siguientes apartados:

- Coste debido a Recursos Humanos
- Coste debido a Recursos Hardware
- Coste debido a Recursos Software
- Otros gastos

Coste debido a Recursos Humanos

En este bloque se muestran los recursos derivados del empleo de personal para la realización de este Proyecto Fin de Carrera. Se ha estimado que se empleará un *Ingeniero Junior* durante 12 meses y un técnico de laboratorio trabajando a tiempo completo para mantener la red y los ordenadores empleados en este trabajo.

De acuerdo con los Baremos de Honorarios Orientativos para Trabajos Profesionales del Colegio Oficial de Ingenieros de Telecomunicación (COIT) para el ejercicio 2006, la tarifa que se debe aplicar a trabajos por tiempo empleado es de $74{,}88 \cdot H_n + 96{,}72 \cdot H_e$ (€), donde H_n representa las horas normales dentro de la jornada laboral y H_e las horas especiales.

Suponiendo que la carga laboral del Ingeniero es de 8 horas al día y de 20 días al mes, sus honorarios mensuales ascienden a 11980,8 €. Teniendo en cuenta que el número total de horas es de 1920, hay que aplicar un coeficiente de corrección de C=0,4 por exceder de las 1080 horas de trabajo.

Para el cálculo del coste de Recursos Humanos relacionada con el mantenimiento de la red y los equipos informáticos hay que tener en cuenta que la red la utilizan 20 personas y que el sueldo base de un Técnico de Mantenimiento es de 900€/mes.

Concepto	Tiempo empleado	Coste mensual	Coste total
Formación	2 meses	11980,8 €	23961,6 €
Desarrollo	8 meses	11980,8 €	95846,4 €
Documentación	1 mes	11980,8 €	11980,8 €
Adaptación de la Memoria a la Normativa de la ETSIT (ULPGC)	1 mes	11980,8 €	11980,8 €
Coste total asociado al Ingeniero Junior			143769,6 €
Coste corregido por superar las 1080 horas ($C_{Final} = C_{Anterior} * 0,4$)			57507,84 €
Técnico de Mantenimiento	12 meses	45 €	540 €
Coste Total			**58047,84 €**

Tabla P 1. Coste de los Recursos Humanos

Coste debido a Recursos *Hardware*

El coste total de los recursos *hardware* se obtiene en función de los usuarios que empleen cada servicio y del tiempo de vida (amortización) de cada herramienta.

Dentro de estos costes se incluye una red de ordenadores personales con impresora y acceso externo. Cada ordenador personal de la red tiene un valor de 1200 € con una amortización de un año, siendo usado por una única persona. Se estima el coste mensual del uso de la impresora de 2 € por usuario de la red. El servidor UNIX sobre el que se realizan las simulaciones del sistema tiene un valor de 2500 €, y da servicio a 20 usuarios, siendo el período de amortización de 18 meses. Para la realización de este Proyecto también se precisa de un ordenador portátil para un solo usuario de valor 1300 €, con un periodo de amortización de 18 meses.

Concepto	Tiempo empleado	Coste mensual	Coste total
Servidor de UNIX	10 meses	7 €	70 €
Ordenador Personal	10 meses	100 €	1000 €
Ordenador Portatil	12 mes	73 €	876 €
Impresora	12 mes	2 €	24 €
Coste Total			**1970 €**

Tabla P 2. Coste de los Recursos Hardware

Coste debido a Recursos *Software*

El coste de los recursos *software* se obtiene a partir del valor de las licencias de los productos utilizados para la realización de este Proyecto Fin de Carrera se muestra en el detalle en la *Tabla P3*. El resto del *software* utilizado es de libre distribución.

Concepto	Tiempo empleado	Coste mensual	Coste total
Microsoft Visual C++	10 meses	*	499 €
Microsoft Windows XP	12 mes	*	89 €
Microsoft Office XP	3 mes	*	389 €
Coste Total			**977 €**

Tabla P 3. Coste de los Recursos Software

Gastos Generales

Además de los gastos presentados en las secciones anteriores hay que añadir un 5% sobre el total de los gastos anteriores para incluir los costes derivados del uso de material necesario para la elaboración del proyecto y del mantenimiento de los laboratorios

Beneficio Industrial

Todo proyecto conlleva un beneficio industrial que justifique la realización del mismo. Para el caso del presente Proyecto, se ha estimado en un 10% de los gastos totales.

En la *Tabla P4* se recopilan los gastos contemplados anteriormente obteniéndose el importe final al que asciende este Proyecto Fin de Carrera incluyendo los impuestos.

Concepto	Importe
Recursos Humanos	58047,84 €
Recursos Hardware	1970 €
Recursos Software	977 €
Otros gastos (5%)	3049,74 €
Coste total	**64044,58 €**
Beneficio Industrial (10%)	6404,45 €
Total	**70449,03 €**
I.G.I.C. (5%)	3522,45 €
Presupuesto Final	**73971,48 €**

Tabla P 4. Coste Total

Dña. M. Teresa Medina León declara que el presupuesto del Proyecto "*Entorno de modelado y análisis rápido de sistemas-en-chip multiprocesador basados en Network-on-Chip (NoC)*" asciende a un total de setenta y tres mil novecientos setenta y un euros con cuarenta y ocho céntimos (73971,48 €).

Fdo.: María Teresa Medina León

En Las Palmas de Gran Canaria, Diciembre de 2006

Bibliografía

[1] R. Bergamaschi, W. R. Lee, “Designing Systems-on-Chip Using Cores“, In Proc of DAC, 2000.

[2] W. Cesário et al, “Component-Based Design Approach for Multicore SoCs,” In Proc. of DAC, 2002.

[3] D. Burger, J. R. Goodman, “Billion-Transistor Architectures: There and Back Again”, In IEEE Computer, March 2004.

[4] S. Naffziger, T. Grutkowski, B. Stackhouse, “The Implementation of a 2-core Multi-Threaded Itanium® Family Processor”, In Proc of ISSCC, 2005

[5] J. Henkel, W. Wolf, S. Chakradhar. “On-chip networks: A scalable, communication-centric embedded system design paradigm”. NEC Laboratories America. Princeton University. Princeton.

[6] K. Keutzer. “System-level design: Orthogonalization of concerns and platform-based design,” In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Dec. 2000.

[7] L. Cai, D. Gajski. “Transaction Level Modeling: An Overview”. In Proceedings of CODES+ISSS’03, California, USA, October 2003.

[8] S. Pasricha, N. Dutt, M. Ben-Romdhane, “Extending the Transaction Level Modeling Approach for Fast Communication Architecture Exploration", In Proc of DAC 2004.

[9] S. Pasricha. ”Transaction level modeling of SoC with SystemC 2.0”, In Synopsys User Group Conference, 2002.

[10] W. Klingauf. “Systematic Transaction Level Modeling of Embedded Systems with SystemC”, *Proc. DATE*, 2005.

[11] T. Grötker, S. Liao, G. Martin, S. Swan. “System Design with SystemC”. Kluwer Academic Publishers, 2002.

[12] F. Ghenassia. “Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems”. Springer, 2005.

[13] A. Rose, S. Swan, J. Pierce, J. Fernandez. “Transaction Level Modeling in SystemC”. SystemC TLM whitepaper, 2005

[14] SystemC Verification Library 1.0, 2003, http://www.systemc.org

[15] Transaction Level Modelling Standard 1.0, June 2005, http://www.systemc.org

[16] Mini-NoC website. http://www.es.ele.tue.nl/~mininoc/

[17] D. A. Patterson, J. L. Hennessy. “Computer Organization and Design: The Hardware/Software Interface”. 3rd Edition. Morgan Kaufmann Publishers.

[18] V. Reyes, W. Kruijzer, T. Bautista, A. Nuñez. "A SystemC based System-Level Design Tool for Multiprocessor SoC Modeling and Simulation", submitted to ACM Transactions on Embedded Computing Systems, 2006

[19] V. Reyes, W. Kruijzer, T. Bautista, G. Marrero, P. Carballo. 2004. CASSE: A System-Level Modeling and Design-Space Exploration Tool for Multiprocessor Systems-on-Chip. In Proceedings of Euromicro Symposium on Digital System Design, Rennes, France, September 2004.

[20] V. Reyes, W. Kruijzer, T. Bautista, G. Marrero, A. Nuñez. 2005. A Multicast Inter-Task Communication Protocol for Embedded Multiprocessor Systems. In Proceedings of CODES+ISSS'05, New York, USA, October 2005.

[21] W. Kruijzer KRUIJTZER, V. Reyes, W. Gehrke. 2006. Design, Synthesis and Verification of a Smart Imaging Core using SystemC. In Design Automation of Embedded Systems (DAES) journal, Springer, 2006.

[22] van de Wolf, P. de Kock E. Hendrikson T., Kruijtzer, W., Essink, G. "Design and Programming of Embedded Multiprocessors: An Interface-Centric Approach". 2004. In Proceedings of CODES+ISSS'04, September 2004.

[23] IEEE 1666TM SystemC standard, 2006, http://www.systemc.org

[24] S. Stuijk. "Design and implementation of a JPEG decoder". Practical Training Report. Technical University of Eindhoven. Eindhoven, The Netherlands. December 2001.

[25] G. Smith. DAC Panel Presentation. 40 th Design Automation Conference (DAC). Anaheim. June 2003.

[26] A. Jantsch, H. Tenhunen. "Networks on Chip". Kluwer Academic Publishers. February 2003.

[27] N. Kavaldjiev, G. J. M. Smit. "A survey of Efficient On-Chip Communications for SoC". University of Twente. Enschede, The Netherlands, 2003.

[28] W.J. Dally, B. Towles. "Principles and Practices of Interconnection Networks". Morgan Kaufmann Publishers. 2004.

[29] A. Jantsch., "Communication Performance in Network-on-Chips". Royal Institute of Technology, Stockholm. November 2003.

[30] L. Cai.. “Comparison of SpecC and SystemC Languages for System Design”. Technical Report, UCI, May 2003.

La última fecha de consulta de las websites referenciadas en esta bibliografía, fue en Agosto de 2006.

Anexo A

Con esta memoria se adjunta, como parte de la documentación del este Proyecto Fin de Carrera, un CD-ROM cuyo contenido es el siguiente:

- En el directorio *Memoria* se encuentra una copia de los capítulos que componen la memoria de este Proyecto Fin de Carrera en formato PDF. En este directorio se encuentra tanto la versión original inglesa como la versión en castellano.

- En el directorio *imágenes* se encuentran las distintas imágenes que se utilizaron para la realización de la memoria, organizadas por capítulos. La mayoría de ellas se encuentran en ficheros en formato *Visio*.

- En el directorio *Mixed-Level Model* se encuentra los ficheros que componen el modelo *Mixed-Level* y las imágenes empleadas en la comparativa. Este directorio se encuentra subdividido a su vez en los siguientes subdirectorios:

    - *comm class: c*ontiene la librería que incluye la implementación de las primitivas de comunicación.
    - *modules:* se encuentra la descripción RTL de cada uno de los módulos que componen la red NOC y la descripción SystemC del módulo adaptador.
    - *JPEG:* contiene los ficheros del Decodificador JPEG con la anotación de la información temporal.
    - *scripts:* se incluyen los ficheros descripción empleados en la herramienta *CASSE*.
    - *test_images:* contiene las distintas imágenes empleadas en la comparativa.

SEGUNDA PARTE:
Memoria