Proyecto de fin de Máster en Ingeniería de Computadores
Curso 2007 - 2008

Un gestor de ejecución de grafos de tareas para sistemas multitarea dinámicamente reconfigurables

Carlos González Calvo

Dirigido por:
Jesús Javier Resano Ezcaray
Dpto: Arquitectura de Computadores y Automática

Facultad de Informática
Universidad Complutense de Madrid

A mi familia y amigos, especialmente
a los que ya no podrán ver el resultado

AGRADECIMIENTOS

Hace unos años, cuando afronté el reto de realizar la Ingeniería en Informática y posteriormente el Máster de Investigación en Informática en Ingeniería de Computadores, me dijeron que era un trabajo personal. Si bien ahora puedo afirmar que, efectivamente, es una labor “básicamente” personal, también he de reconocer que, sin la ayuda y los apoyos recibidos, este largo camino se habría convertido en una empresa imposible. Debo agradecer:

A mi Director de proyecto, Javier. Realmente, tu labor en este camino comenzó hace dos años, cuando yo era un alumno más en búsqueda de un proyecto fin de carrera dentro del Departamento de Arquitectura de Computadores y Automática. Entonces, gracias a tu pasión por la ingeniería y tu actitud de ayuda sin condiciones, sembraste un ingeniero entusiasta de su trabajo y dispuesto a llegar siempre, un poco más lejos. Gracias por la confianza que has puesto en mí.

A los “compañeros” del grupo de investigación, esperemos que dentro de poco se puedan quitar las comillas. Sabía que me incorporaba a trabajar con excelentes profesionales, pero lo que es más importante para mí, con magníficas personas:

A Daniel. Debo agradecerte especialmente la ayuda que me has prestado estos años. Las correcciones exhaustivas de las memorias, los acertados análisis técnicos sobre nuestras publicaciones y la aportación de tu experiencia. Tu visión crítica y humana de la vida y de la ingeniería, junto con tu capacidad de dar ánimos en los momentos más difíciles, me han ayudado y me siguen ayudando a afrontar los distintos retos.

A Juanan. He tenido la gran suerte de haber compartido este largo camino contigo y haberme podido contagiar de tu espíritu de superación, sacrificio y esfuerzo; que aún siendo todos estos aspectos inestimables, no son más que un apéndice comparado con tu amistad.

A mis amigos, que sabéis apreciar las aventuras en las que me embarco (como esta memoria), en especial a Adolfo. Perdonarme si este último año no os he dedicado todo el tiempo que merecíais.

A mi familia. Hemos avanzado estos años contra viento y marea, sin vuestra fortaleza y cariño, nada de esto hubiera sido posible. Especialmente a mis padres y hermano por el apoyo que me brindaron, por la formación, por fomentar en mí el deseo de saber, de conocer lo novedoso y abrirme las puertas al mundo ante mi curiosidad insaciable. Esto es también vuestro premio.

Gracias a todos.

ÍNDICE

ÍNDICE DE IMÁGENES Y TABLAS

RESUMEN DEL PROYECTO Y PALABRAS CLAVE

RESUMEN DEL PROYECTO

El HW reconfigurable se puede utilizar para construir un sistema multitarea en el que las tareas puedan asignarse en tiempo de ejecución a los recursos reconfigurables según las necesidades de las aplicaciones. En estos sistemas, las tareas se representan normalmente como grafos de subtareas acíclicos, donde una subtarea es la unidad de planificación. Normalmente, un procesador empotrado controla la ejecución de este tipo de sistemas trabajando con estructuras de datos complejas, como grafos o listas enlazadas, cuyo manejo a menudo genera retardos en la ejecución. Además, las comunicaciones HW/SW son a menudo un cuello de botella del sistema. Por tanto resulta muy interesante reducir tanto los cálculos que realiza el procesador como las comunicaciones. Para lograr este objetivo se ha desarrollado un gestor HW que controla la ejecución de grafos de subtareas en un conjunto de unidades reconfigurables. Este gestor recibe como entrada los grafos junto con una planificación asociada a cada subtarea y garantiza su correcta ejecución sin necesidad de ninguna otra intervención por parte del procesador. Además, incluye mecanismos para optimizar la gestión de las reconfiguraciones reduciendo las penalizaciones que generan en tiempo de ejecución.

ABSTRACT

Reconfigurable HW can be used to build a hardware multitasking system where tasks can be assigned to the reconfigurable HW at run-time according to the requirements of the running applications. Normally the execution in this kind of systems is controlled by an embedded processor. In these systems tasks are frequently represented as acyclic subtask graphs, where a subtask is the basic scheduling unit that can be assigned to a reconfigurable HW. In order to control the execution of these tasks, the processor must manage at run-time complex data structures, like graphs or linked list, which may generate significant execution-time penalties. In addition, HW/SW communications are frequently a system bottleneck. Hence, it is very interesting to find a way to reduce the run-time SW computations and the HW/SW communications. To this end I have developed a HW execution manager that controls the execution of subtask graphs over a set of reconfigurable units. This manager receives as input a subtask graph coupled to a subtask schedule, and guarantees its proper execution. In addition it includes support to reduce the execution-time overhead due to reconfigurations. With this HW support the execution of task graphs can be managed efficiently generating only very small run-time penalties.

PALABRAS CLAVE

Hardware multitarea, reconfiguración parcial dinámica, gestor de tareas, SoC, unidad reconfigurable (UR), tabla asociativa, FPGA, ISE, EDK, Virtex II-PRO.

AUTORIZACIÓN

El ponente Carlos González Calvo, con DNI 77811220, autoriza a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos tanto la propia memoria, como el código, la documentación y el prototipo desarrollado.

Carlos González Calvo

En Madrid, a 23 de junio de 2008

INTRODUCCIÓN

La reconfiguración parcial [1] abre la posibilidad de desarrollar un sistema multitarea HW dividiendo la totalidad del área reconfigurable en unidades reconfigurables (URs) que incluyan un interfaz fijo conocido por el diseñador de tareas [2]. Estas URs se pueden reconfigurar para ejecutar tareas de las aplicaciones que se estén ejecutando; por tanto, con las configuraciones correctas, los mismos recursos se pueden personalizar en tiempo de ejecución para acelerar varias aplicaciones específicas, o para adaptarse a los requisitos variables de una aplicación dinámica compleja. Además, las reconfiguraciones se pueden actualizar en tiempo de ejecución para extender la funcionalidad del sistema, mejorar las prestaciones o arreglar errores. Actualmente, algunas plataformas comerciales, como XILINX Virtex™ series [3] o Altera® Stratic [4], se pueden utilizar para implementar un sistema multitarea HW en el que varias URs colaboren con uno o más procesadores empotrados. Es más, algunos vendedores han desarrollado entornos de diseño especiales para proporcionar soporte a este tipo de sistemas, como el XILINX Embedded Development Kit (EDK) [5] o de Altera SOPC builder [6].

En sistemas empotrados, las tareas se representan frecuentemente como grafos acíclicos y lo más común es que un procesador empotrado deba encargarse de su ejecución teniendo en cuenta sus dependencias internas. Esto implica tratar con estructuras de datos complejas y realizar cuantiosas comunicaciones HW/SW. Debido a que el trabajo de este gestor es muy crítico para el sistema, es esencial evaluar su eficiencia y explorar implementaciones alternativas. Además, este gestor puede mejorar en gran medida su rendimiento implementando algunas estrategias de optimización simples. No obstante, desarrollar gestores específicos para cada grafo de tareas es un proceso complejo y propenso a errores. Por tanto, desarrollar un gestor genérico que pueda tratar con cualquier grafo de tareas puede simplificar enormemente el proceso de diseño.

En este proyecto he diseñado e implementado un gestor genérico que recibe información de un planificador de tareas y garantiza la correcta ejecución de un grafo de tareas considerando tanto las dependencias internas como la planificación seleccionada. Además, el gestor identifica qué subtareas pueden ser reutilizadas a partir de ejecuciones previas y proporciona soporte para aplicar una técnica de precarga para ocultar las latencias de reconfiguración [7].

CONTEXTO DE LA MEMORIA

El trabajo de investigación que ha dado como resultado esta memoria se ha desarrollado en el Departamento de Arquitectura de Computadores y Automática de la Universidad Complutense de Madrid. Este departamento está compuesto por los siguientes grupos de investigación:

- ***Arquitectura de sistemas distribuidos****:* El grupo DSA lleva a cabo varias actividades de investiación relacionadas con la asignación dinámica y la planificación de tareas (GridWay Metascheduler), la penalización de máquinas virtuales respecto a recursos grid, gestión dinámica de máquinas virtuales (OpenNebula Virtual Infrastructure Engine) y federación de infraestructuras grid (Grid4Utility).

- ***Arquitectura y tecnología de sistemas informáticos***: La actividad investigadora del Grupo de Arquitectura y Tecnología de Sistemas Informáticos (ArTeCS) de la Universidad Complutense de Madrid se centra en la concepción y construción de sistemas informáticos digitales, y su eficiencia en cuanto a rendimiento, consumo de energía y coste. Dentro de esta amplia área, el grupo pestra mayor atención a procesadores de alto rendimiento, computación distibuida, sistemas empotrados y systems-on-chip.

- ***Ingeniería de sistemas, control, automatización y robótica*** (ISCAR): El grupo centra su investigación en las siguientes líneas: Estabilización de barcos rápidos, Robótica terrestre y aérea , Visión por computador, Avionica y temas del espacio, Algoritmos Genéticos, Logística y optimización, Sistemas Borrosos, Robots marinos cooperantes,E-learning y laboratorios virtuales, Simulación y Control, modelado y optimización de procesos.

- ***Grupo de Hardware Dinámicamente Reconfigurable****:* Grupo de investigadores que vienen del campo de Síntesis de Alto Nivel y co-diseño HW/SW. Actualmente, centran su interés en la gestión de recursos de hardware dinámicamente reconfigurable, en sistemas de propósito general que ejecutan una serie continua de tareas. El problema de la gestión de hardware para sistemas hardware multitarea es similar al problema de gestión de memoria y tiempo de CPU para sistemas software multitarea.

El trabajo de investigación presentado en esta memoria, se enmarca en esta última línea de investigación, *Hardware Dinámicamente Reconfigurable*.

Básicamente durante los años 80 y principio de los noventa, los circuitos reconfigurables más utilizados fueron PALs y CPLDs. Éstos son dispositivos reconfigurables de baja capacidad. La función que debe realizar el circuito se diseña y modifica en el ordenador del diseñador. La configuración resultante se descarga en el propio laboratorio con unos medios sencillos. Este mecanismo supuso una clara ventaja para el diseño de prototipos y pequeñas series respecto a la utilización de componentes discretos (circuitos integrados digitales estándar).

La utilidad de estos dispositivos de baja capacidad no se centró únicamente en el prototipado, sino que también se extendió a diseños industriales, básicamente a aplicaciones de *glue logic*. La utilización de éstos permitía la agrupación de componentes discretos con la consecuente reducción de área. Además, dotaban de una cierta flexibilidad para corrección de errores y actualización del sistema sin necesidad de cambio en el circuito impreso, compensando de esta forma el sobrecoste de estos dispositivos respecto a la utilización de circuitos integrados discretos.

A principio de los años 90 hacen su entrada en el mercado de los dispositivos reconfigurables las FPGAs con tecnología SRAM. Éstas disponían de una capacidad lógica mayor que las CPLDs y PALs. En los prototipos del grupo ya se utilizaban FPGAs encargadas de realizar procesamientos más complejos que un simple *glue logic*, básicamente, preprocesamiento para comunicaciones digitales y algoritmos para visión artificial.

Sin embargo, la evolución de las FPGAs no acaba ahí. Al igual que ocurre con los circuitos de aplicación específica (*Application Specific Integrated Circuit* -ASIC-), la capacidad de integración en los mismos se incrementa de forma sorprendente año a año. La conocida Ley de Moore [8] que preveía que el número de transistores integrables en una misma área de silicio se duplicaría cada 18 meses se está cumpliendo.

Esta tendencia, que ha permitido que en la actualidad se integren sistemas digitales completos en ASICs surgiendo los *System-on-Chips* (SoCs), también afecta a los dispositivos reconfigurables, en concreto a las FPGAs. En la actualidad se dispone de FPGAs con capacidades de integración de millones de *puertas lógicas equivalentes* [9]. Esta situación permite, al igual que con los ASICs, integrar un sistema digital completo en un único dispositivo FPGA, definiéndose este caso como *System-on-Programmable-Chip* (SoPC).

Frente a los SoCs basados en tecnología ASIC, los SoPCs tienen una desventaja en cuanto a precio y a eficiencia puesto que no se tratan de circuitos diseñados específicamente para una determinada aplicación. Sin embargo, en muchos casos compensa esta situación con su flexibilidad y accesibilidad para los grupos de investigación y para las pequeñas y medianas empresas.

La necesidad de gestionar la complejidad generada por este nivel de integración en los SoCs y SoPCs ha incorporado nuevas metodologías basadas en el diseño a partir de módulos

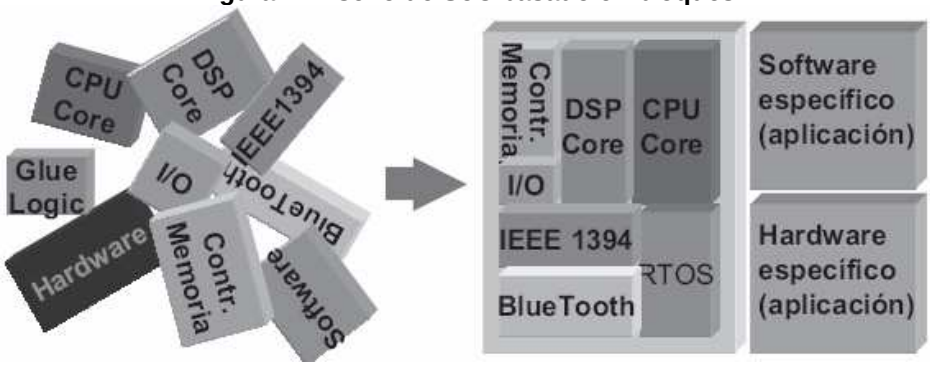
hardware y módulos software. Sobre la base de este concepto, se define un SoC de la siguiente forma: "Un sistema *System-on-a-Chip* (SoC) está compuesto por un conjunto de módulos y subsistemas (hardware y software) que se interconectan de forma apropiada para cumplir las funciones requeridas en una determinada aplicación" [10]. R. Rajsuman también define el concepto de SoC como "un circuito integrado, diseñado mediante interconexión de múltiples bloques VLSI independientes, que ofrece toda la funcionalidad de una aplicación" [11]. Esta definición hace hincapié en los bloques previamente diseñados que agrupan funcionalidades complejas. J. Borel define el mismo término como "la agrupación en un único *chip* de las capacidades de procesamiento, control, almacenamiento, comunicación, sensórica y de actuación" [12]. Los dispositivos actuales no permiten la integración de todas estas capacidades, pero sí de una gran parte de ellas, siendo necesario en la mayoría de ocasiones el uso de memoria externa o de algún periférico.

El diseño de un SoC se basa en la reutilización de componentes prediseñados, denominados IP-Cores, *cores* o módulos, reutilizando incluso arquitecturas predefinidas. Aún así, la complejidad del diseño de SoCs es mayor respecto a la del diseño convencional de sistemas distribuidos en circuitos impresos y en distintos circuitos integrados, pero aporta numerosas mejoras como son [10]:

- Reducción del coste del producto.
- Incremento del rendimiento del sistema. Al integrar los buses en un dispositivo se disminuyen las líneas de interconexión existentes en el circuito impreso y además se reduce considerablemente la longitud de las mismas lo que permite velocidades mucho mayores de comunicación.
- Descenso significativo del consumo.
- Reducción global de la superficie de silicio.

El diseño de un sistema SoC implica integrar en un único dispositivo un sistema electrónico completo mediante diversos bloques complejos (figura 1). Estos bloques son tanto bloques hardware (microprocesadores, procesadores digitales de señal, buses *on-chip*, periféricos, etc.) como bloques software. Los elementos software pueden abarcar desde sistemas operativos en tiempo real a programas especificos para la aplicación.

Figura 1. Diseño de SoC basado en bloques



En cuanto a los *cores* hardware, se diferencian tres tipos en función de su flexibilidad para la implementación:

- **Cores *soft***: Estos módulos están descritos en lenguajes de descripción hardware (VHDL o Verilog) a nivel de transferencia entre registros (*Register-Transfer-Level* RTL). El código representa entidades hardware sintetizables.

  Este tipo de *cores* son flexibles y relativamente independientes de la tecnología, pudiéndose sintetizar para distintos dispositivos, incluso para ASICs o lógica programable.

- **Cores *firm***: La descripción de los mismos se realiza mediante información a nivel de puertas y biestables (*netlist*) optimizada en tamaño y rendimiento para un dispositivo o familias de dispositivos en concreto. Por tanto la asignación de recursos del dispositivo para realizar esas tareas ya está realizada antes de su utilización por parte del diseñador, restando únicamente la fase de interconexión que se realizará en la implementación.

  Son menos flexibles pero permiten la protección de la Propiedad Intelectual y son más eficientes en general.

- **Cores *hard***: En el caso del diseño de ASICs este tipo de bloques son *cores* preparados para ser integrados en el silicio del dispositivo.

  Disponen además de la información de los recursos necesarios a bajo nivel que hay que implementar en el silicio, de la información del rutado exacto. Están optimizados en aspectos tales como el rendimiento, área o consumo.

  En el caso de los *cores hard* para dispositivos lógicos programables, son *cores* ya integrados en el silicio del dispositivo.

Los *cores soft* son los que admiten mayor grado de flexibilidad mientras que los *cores hard* son los que resuelven las necesidades más complejas (procesadores avanzados, multiplicadores de alto rendimiento, etc.) y se encuentran mejor optimizados.

El diseño de un SoC agrupa y mezcla las disciplinas del diseño hardware, diseño software y telecomunicaciones. Este nuevo concepto de diseño crea un un entorno convergente que abre unas dimensiones innovadoras de productos, mercados, competidores, tecnologías y servicios [13]. Los participantes en este entorno son los fabricantes de IP-Cores y de circuitos integrados; los diseñadores y verificadores de sistemas; los desarrolladores de sistemas operativos y de aplicaciones; así como las empresas dedicadas a las herramientas de desarrollo que permitan esta evolución.

Para que estos *cores* sean realmente reutilizables, deben presentar interfaces comunes que permitan una rápida y sencilla integración. Con este propósito se están estableciendo consorcios e iniciativas. Un ejemplo significativo es el del consorcio *Virtual Socket Interface Aliance (VSIA)* [14] cuyo objetivo es el de normalizar el desarrollo de *cores*. Esta evolución en la forma de diseñar los sistemas electrónicos ofrece numerosos grados de libertad al diseñador permitiendo no sólo la elección de los *cores*, sino también la selección de la topología de bus y de la arquitectura del sistema.

En la práctica no son manejables tantos grados de libertad. Con el fin de guiar al diseñador en la tarea de seleccionar una arquitectura para su sistema se ha evolucionado hacia un modo de diseño con *cores* más restringido denominado *diseño basado en plataformas* [15].

En este caso, el fabricante de los dispositivos propone al diseñador el uso de una especificación concreta para la interconexión de los *cores* y unas determinadas topologías de bus. Generalmente, estas propuestas están condicionadas por el microprocesador que integre el fabricante en los dispositivos. Los fabricantes de lógica programable proponen sus plataformas para desarrollar sistemas integrados en un *chip* empleando sus FPGAs de alta capacidad.

Estas plataformas SoPC están resultando un rotundo éxito dentro del área de los SoCs [15]. En la actualidad, sólo un pequeño grupo de grandes multinacionales se aventura a diseñar SoC sobre ASICs. En contraste, con los SoPCs basados en plataformas han puesto al alcance de toda la comunidad diseñadora de sistemas electrónicos la posibilidad de realizar diseños integrando sus desarrollos específicos y reutilizando los ya realizados por otras fuentes en un único dispositivo.

Dada la rápida evolución de estos sistemas y la revolución producida en las metodologías de diseño, se ha considerado necesario en el departamento un grupo de investigación dedicado al hardware dinámicamente reconfigurable en que se enmarca esta memoria. Mediante esta línea de investigación, además de mantener un seguimiento de los avances en esta tecnología, se realizan investigaciones en aspectos innovadores de estos sistemas, sirviendo de ejemplo el trabajo de investigación presentado en esta memoria.

TRABAJO RELACIONADO

Recientemente muchos grupos de investigación han propuesto crear sistemas multitarea HW usando recursos parcialmente reconfigurables. Marescaux et al. presentaron en [2] la primera implementación de un sistema de esas características en una FPGA comercial que estaba particionada en un conjunto de URs idénticas. Otras aproximaciones recientes que presentan sistemas HW multitarea son [16], [17] y [18]. En estos sistemas, normalmente las URs están fuertemente acopladas a un procesador que guía la ejecución del sistema. Este procesador debe monitorizar la ejecución HW y encargarse tanto del Sistema Operativo (SO) como de las gestiones de los grafos en ejecución. Adicionalmente, este procesador podría tener que ejecutar también otras tareas; con lo que, si estaba demasiado ocupado gestionando la ejecución de las URs, se producirían retardos importantes no sólo en la ejecución de sus propias tareas, sino que también en todo el sistema. Una forma de reducir el impacto de este problema es distribuir el SO y las computaciones de las planificaciones entre los elementos de proceso. Esta aproximación reduce tanto la carga computacional del procesador como las costosas comunicaciones HW/SW. Algunos ejemplos interesantes de sistemas HW multitarea son [19], [20] y [21]. En [19] los autores proponen un SO distribuido con soporte para comunicaciones entre tareas. En [20] los autores proponen un planificador dinámico basado en HW para arquitecturas reconfigurables que aplica una heurística de planificación en listas. Sin embargo, los autores no implementaron su diseño, sino que sólo la incluyeron en su entorno de simulación específico. Finalmente en [21] los autores proponen incluir módulos HW distribuidos que colaboran con un procesador para soportar llamadas de sistema operativo de tipo UNIX para tareas HW.

CONTRIBUCIÓN DEL PROYECTO

La principal contribución de esta memoria es la descripción de un gestor de ejecución de grafos de tareas para sistemas multitarea dinámicamente reconfigurables. Este gestor controla la ejecución de una serie de subtareas en un conjunto de unidades reconfigurables siguiendo un grafo de subtareas dado y una planificación. Esta información se almacena en una tabla asociativa y un conjunto de FIFOs respectivamente.

Además, ya que la latencia de reconfiguración puede generar retardos significativos en la ejecución (para las FPGAs estos retardos son del orden de milisegundos [22]), el gestor aplica dos técnicas para reducir estas penalizaciones. En primer lugar, antes de reconfigurar una UR para cargar una subtarea en ella, el gestor comprueba si esa subtarea había sido cargada allí previamente. Si esto ocurriese, la configuración puede ser reutilizada directamente sin necesidad de cargarla. En segundo lugar, el gestor aplica una técnica de anticipación [23] para ocultar las latencias de reconfiguración en la medida de lo posible.

Dado que, planificar las reconfiguraciones es algo crítico para un sistema HW multitarea, y muchos autores han propuesto algunas heurísticas específicas que consiguen resultados casi óptimos [24]. Para proporcionar soporte a estos planificadores, el gestor se puede programar para que siga una secuencia de reconfiguraciones dada (aproximación *basada en una planificación*). En este caso, el gestor identifica cuál es la tarea que debe reconfigurarse a continuación de acuerdo a la planificación dada (que está almacenada en una FIFO), y comienza su reconfiguración tan pronto como sea posible.

OBJETIVOS Y LOGROS REALIZADOS

Los objetivos iniciales del proyecto incluían el diseño del gestor, su implementación en VHDL o verilog, su ejecución sobre una placa de FPGA y la realización de una comparativa con un gestor SW equivalente.

Tras tener claro una visión global del sistema, se repartió el gestor en diferentes unidades quedando fijadas las tareas que debía realizar cada una y las interacciones entre ellas. En el proceso de desarrollo del proyecto, se simultaneó el diseño de una nueva unidad con la implementación y testeo en ISE de la unidad anteriormente diseñada, lo que permitió tener una primera versión en relativamente poco tiempo. El lenguaje escogido para la implementación del gestor fue VHDL ya que, era el lenguaje más conocido por mí.

Para comprobar la correcta funcionalidad de los diseños realizados, el área que ocupan y su rendimiento, todos los módulos fueron sintetizados y comprobados desarrollando bancos de pruebas para las herramientas de simulación Post Place & Route en el Ise 9.1i. Estas simulaciones garantizan el mismo resultado que una ejecución en la FPGA final, pero resulta mucho más cómodo trabajar con ellas al poder definirse de forma sencilla todo tipo de bancos de pruebas.

Finalmente, se realizó una comparativa con un gestor SW equivalente implementado en C y que se ejecutó en un POWERPC y en MicroBlaze con ayuda de la herramienta EDK 9.1. Los resultados obtenidos nos confirmaron la rapidez del sistema HW frente al SW.

ESTRUCTURA DEL DOCUMENTO

El presente documento está dividido en dos partes bien diferenciadas. En la primera se realiza una aproximación tecnológica de todos los dispositivos, técnicas de reconfiguración y software involucrados en la realización del proyecto. Se describen en detalle las características y funcionalidades de la FPGA, y las herramientas de diseño empleadas para la realización del proyecto: *Xilinx ISE 9.1i, ModelSim 6.0a* y *Xilinx EDK 9.1i*. Finalmente, se ofrece una visión general del HW reconfigurable, sus alternativas de diseño y cómo y por qué es interesante utilizalo para implementar un sistema con varias unidades reconfigurables.

En la segunda parte de la memoria, se describe todo el diseño del proyecto. Se explican todas las unidades en que se divide el gestor: describimos las FIFOs utilizadas, la tabla de tareas (entrada de la tabla, red iterativa, HW de control y finalmente, su esquemático a alto nivel), los módulos que caracterizan a las unidades reconfigurables, las FIFOs de reconfiguraciones y de eventos, el árbitro que controla las escrituras en la FIFO de eventos y la unidad de control. Asimismo, para comprender mejor el funcionamiento del sistema, explicamos la ejecución de un grafo de subtareas paso a paso. Por otro lado, se realiza un análisis comparativo para conocer las ventajas que proporciona este diseño hardware sobre un control meramente software. Finalmente, exponemos algunas conclusiones y posibles ampliaciones del proyecto que se pueden realizar como trabajo futuro.

Como apéndice, se muestra el código VHDL desarrollado, el pseudocódigo C utilizado para la versión software comparativa y un glosario de términos.

APROXIMACIÓN TECNOLÓGICA

SISTEMAS RECONFIGURABLES

Introducción

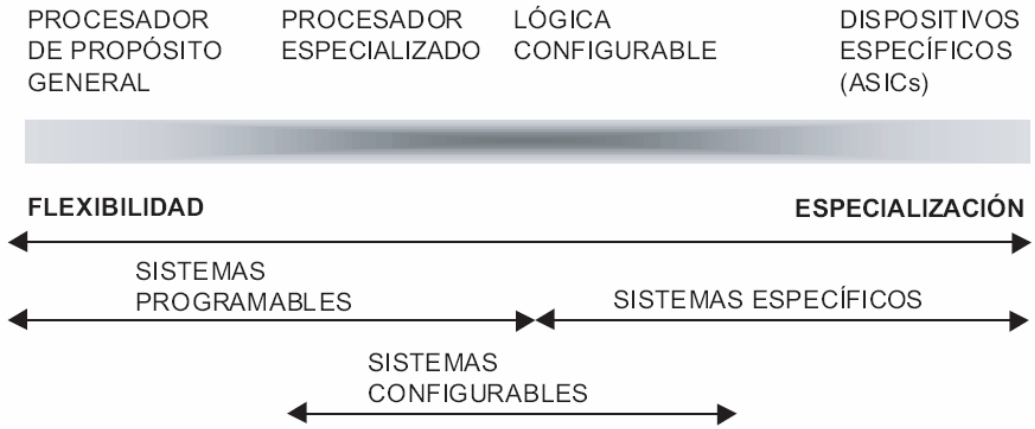
Los métodos convencionales para realizar computación son dos. El primero de ellos consiste en el procesamiento hardware utilizando circuitos cableados de forma fija, bien mediante la integración en un circuito de aplicación específica (Application Specific Integrated Circuit -ASIC-) o bien conectando componentes individuales en una placa [25]. El segundo método, denominado procesamiento software, se basa en el uso de microprocesadores que ejecutan un conjunto de instrucciones para realizar la computación.

El primer sistema se caracteriza por su rapidez y eficiencia para la aplicación concreta para la que ha sido diseñado, pero el circuito no puede ser alterado después de la fabricación, lo que le resta flexibilidad. Usando un microprocesador se incrementa la flexibilidad del sistema para poder cambiar la funcionalidad empleando otro software, pero se reduce la eficiencia debido a las secuencias necesarias de lectura, decodificación y ejecución de instrucciones.

Los dispositivos reconfigurables vienen a cubrir el espacio existente entre estos dos métodos, de forma que se disponga de la eficiencia del procesamiento hardware y de un alto grado de flexibilidad [26]. La adaptabilidad de las arquitecturas reconfigurables permite explotar el paralelismo existente en muchas aplicaciones de forma que se realice computación específica. En la figura 2 se muestra cómo los sistemas configurables, basados en dispositivos reconfigurables, se sitúan en una zona intermedia en la relación flexibilidad-especialización. No son tan flexibles como un procesador de propósito general ni tan específicos, ni tan óptimos, como un ASIC. Sin embargo se benefician de las características positivas de ambos.

Las diferencias más importantes entre la lógica reconfigurable y el procesamiento convencional se pueden resumir en los siguientes aspectos según K. Bondalapati y V. K. Prasanna [27]:

Figura 2. Sistemas de computación según el grado de flexibilidad



- **Computación espacial**: El procesamiento de los datos se realiza distribuyendo las computaciones de forma espacial, en contraste con el procesamiento secuencial.

- **Ruta de datos (datapath) configurable**: Empleando un mecanismo de configuración es posible cambiar la funcionalidad de las unidades de computación y de la red de

interconexión.

- **Control distribuido**: Las unidades de computación procesan datos de forma local en vez de estar gobernados por una única instrucción.

- **Recursos distribuidos**: Los elementos requeridos para la computación se encuentran distribuidos por todo el dispositivo, en contraste con una única localización.

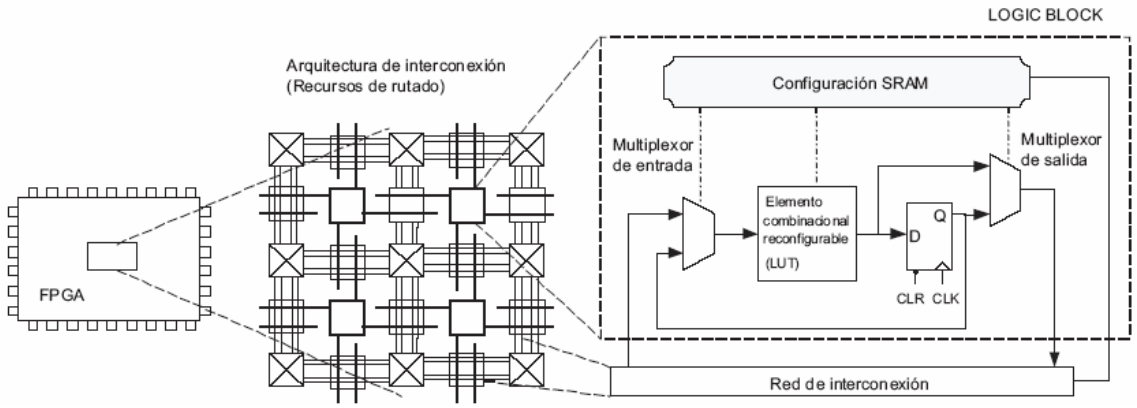
Otras características importantes de estos sistemas destacadas por A. DeHon [28] y R. Tessier [29] son la capacidad de adaptación, la posibilidad de configuración en tiempo de ejecución y la especialización. Beneficiándose de estas características específicas se han desarrollado sistemas reconfigurables eficientes para aplicaciones como la programación genética [30], detección de patrones de texto [31, 32], criptografía [33, 34, 35], compresión de datos [36] o procesamiento de imágenes [37, 38] entre otras.

En este apartado se realizará un repaso de varias arquitecturas reconfigurables que han dado soporte al desarrollo de la disciplina de la Computación Reconfigurable. A lo largo del mismo se presenta la evolución sufrida desde los primeros dispositivos FPGAs hasta las plataformas híbridas que integran microprocesadores de propósito general y ASICs en el mismo chip entre otros elementos. De esta forma la distribución de la computación puede ser repartida entre los distintos componentes del sistema. También se ha incluido una sección donde se presentan los distintos modos de configuración de los dispositivos reconfigurables, siendo estas distintas modalidades uno de los factores más característicos de los mismos.

Tecnología Reconfigurable: FPGAs

Las FPGAs consisten en una matriz de bloques lógicos (Logic Blocks -LBs-) y una red de interconexión. La funcionalidad de los LBs y las conexiones de la red de interconexión pueden modificarse mediante la descarga de los bits de configuración en el hardware [27]. La configuración del dispositivo se realiza empleando dispositivos anti-fuse [39] o bits de memoria SRAM que controlan la configuración de los transistores [40]. El primer modo de configuración tiene menor capacidad de reprogramación mientras que la configuración mediante elementos de memoria SRAM es más versátil admitiendo incluso reconfiguración dinámica y/o parcial.

Figura 3. Modelo genérico de una FPGA



La figura 3 muestra la estructura interna simplificada de una FPGA. A modo ilustrativo se ha elegido para la representación una distribución de tipo isla empleada en varias familias de Xilinx. Existen otras arquitecturas de interconexión como la basada en filas [41], sea-of-gates [42], jerárquica o estructuras en una única dimensión como las empleadas en Garp [33], Chimaera [43] o NAPA [44].

Los LBs interconectados mediante esa red contienen típicamente un circuito combinacional

programable Look-Up Table (LUT), un biestable, lógica adicional y las celdas de memoria SRAM requeridas para la configuración de todos los elementos. Las tareas de entrada-salida se realizan en la periferia del dispositivo, bien mediante LBs o disponiendo de bloques específicos denominados Input-Output-Blocks (IOBs). Actualmente se integran habitualmente otros elementos como son los bloques de memoria RAM dedicada [45], multiplicadores e incluso microprocesadores.

Una de las clasificaciones más habituales del HW reconfigurable se realiza atendiendo a su granularidad. La granularidad de la lógica reconfigurable se define como el tamaño de la menor unidad funcional que es tratada por las herramientas de emplazamiento y rutado [27]. Las FPGAs de grano fino disponen de elementos funcionales de pequeño tamaño, lo que las dota de una mayor flexibilidad. Sin embargo, sufren de retardos elevados cuando se componen circuitos complejos. De forma típica son unidades funcionales de 2 a 4 entradas. El HW reconfigurable con unidades funcionales grandes se denominan de grano grueso, existiendo arquitecturas como la Chameleon [46] con elementos aritméticos de 32 bits o la Morphosys compuesta por un componente reconfigurable, un procesador RISC y una interfaz de memoria con un gran ancho de banda.

Con la integración de múltiples elementos arquitecturales (entendidos como procesadores, memoria e interfaces para periféricos) en FPGAs que disponen de una sección de lógica programable por el usuario surge el concepto de Arquitecturas Híbridas [27].

El avance tecnológico que ha permitido la integración de sistemas en un único dispositivo ha ido acompañado de diversa terminología, todavía no normalizada, para designar a estos sistemas. A continuación se detallan algunos de estos términos:

- **System-on-Chip (SoC)**: Circuito integrado formado por diversos módulos VLSI con distinta funcionalidad que interconectados entre sí ofrecen toda o casi toda la funcionalidad específica para una aplicación.

- **System-on-Programable-Chip (SoPC)**: Se aplica este término específicamente cuando el dispositivo utilizado para realizar el sistema en un chip es reconfigurable. En los SoPC no se utiliza la capacidad de reconfiguración dinámica que puedan disponer estos integrados, sino únicamente las facilidades que ofrecen estos dispositivos en la fase de desarrollo y posteriores actualizaciones del sistema.

- **Configurable-System-On-Programable-Chip (CSoPC)**: Mediante este término se definen los sistemas SoPC en los que se hace uso de la capacidad de reconfiguración de los mismos para aplicaciones de Computación Reconfigurable. Pueden incluirse bajo la denominación CSoPC tanto los sistemas que admiten diferente configuración estática según ciertas condicionantes, como los que utilizan la reconfiguración parcial dinámica para modificar en tiempo de ejecución una sección hardware.

- **Multiprocessor-Configurable-System-On-Programable-Chip (MCSoPC)**: Se aplica esta definición a los sistemas CSoPC que incluyen varios procesadores que ejecutan software, funcionando de forma simultánea.

Dentro de las distintas arquitecturas dinámicamente reconfigurables actuales, las FPGAs dominan ampliamente el mercado. La razón principal es que se basan en una tecnología madura con muchos años de desarrollo y sustentada por un amplio conjunto de herramientas de diseño. Por esta razón, el trabajo realizado en este proyecto ha tomado a las FPGAs como punto de referencia a la hora de desarrollar los prototipos y evaluar los resultados.

Tipos de configuración

De forma general un dispositivo reconfigurable se configura cargando en el mismo una secuencia de bits denominada bitstream. El modo de carga varía según la interfaz que éste disponga. Las interfaces de configuración son de tipo serie o paralelo. El tiempo de

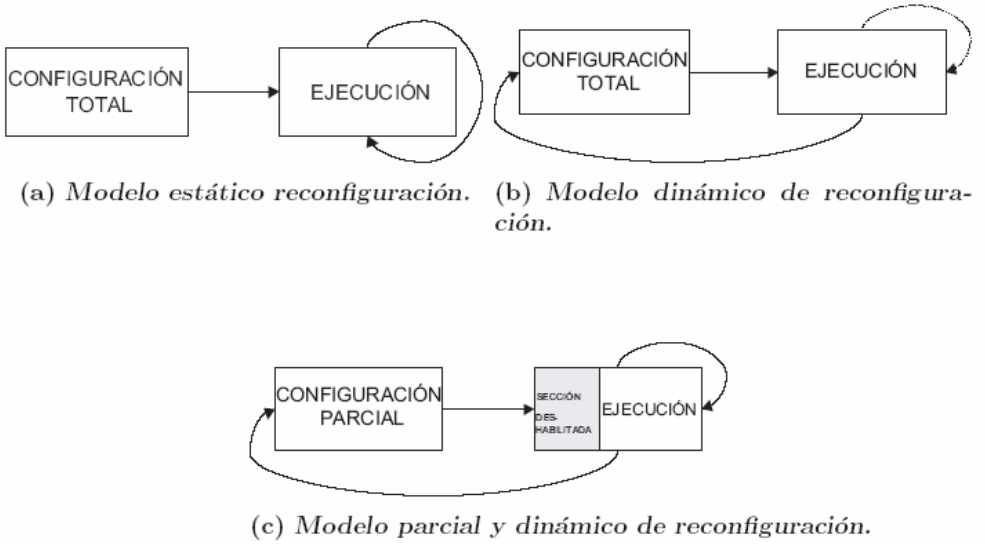
configuración es directamente proporcional al tamaño del bitstream. Las FPGAs de grano fino tienen, en general, tiempos de configuración mayores que las de grano grueso debido al mayor tamaño de sus bitstreams. Esto es debido a que tienen muchos elementos para ser configurados lo que implica ficheros de configuración grandes.

Las FPGAs tradicionalmente se han utilizado para realizar una determinada función en un único contexto, realizándose una configuración de todo el dispositivo. En el caso de que se desee reconfigurar en tiempo de funcionamiento, la reconfiguración de todo el dispositivo es un proceso lento y limitado. La lógica que se va a reconfigurar debe parar la computación y continuar tras la nueva configuración. La penalización impuesta por el tiempo de reconfiguración es importante [47], haciendo en muchas aplicaciones inviable la aplicación de la reconfiguración. A continuación se presentan brevemente los modelos de reconfiguración más representativos:

- **Reconfiguración estática**: Implica parar el sistema y reiniciarlo con una nueva configuración. Su utilidad se centra en los procesos de diseño (depuración) y en la actualización de sistemas. Cada aplicación dispone de una configuración que se carga una vez tras el encendido. La mayoría de los sistemas realizados en la actualidad con lógica reconfigurable disponen de este tipo de reconfiguración, también denominada reconfiguración en tiempo de compilación (en el proceso de diseño). La figura 4(a) sintetiza este modo de operación en el que tras la configuración comienza la ejecución de la lógica configurada sin posibilidad de una nueva carga.

- **Reconfiguración dinámica**: Con el objetivo de obtener un equilibrio entre velocidad de ejecución y área de silicio surge el modo de configuración dinámico mediante el cual se modifica la lógica configurable (incluyendo el rutado) en tiempo de ejecución. La reconfiguración dinámica se basa en el concepto de Hardware Virtual [48] de forma similar a la memoria virtual. Utilizando la capacidad de reprogramación del dispositivo se cambian las configuraciones según se requieren distintas computaciones, reduciendo de esta manera el área de circuito necesaria.

Figura 4. Modelos de configuración



(a) *Modelo estático reconfiguración.* (b) *Modelo dinámico de reconfiguración.*

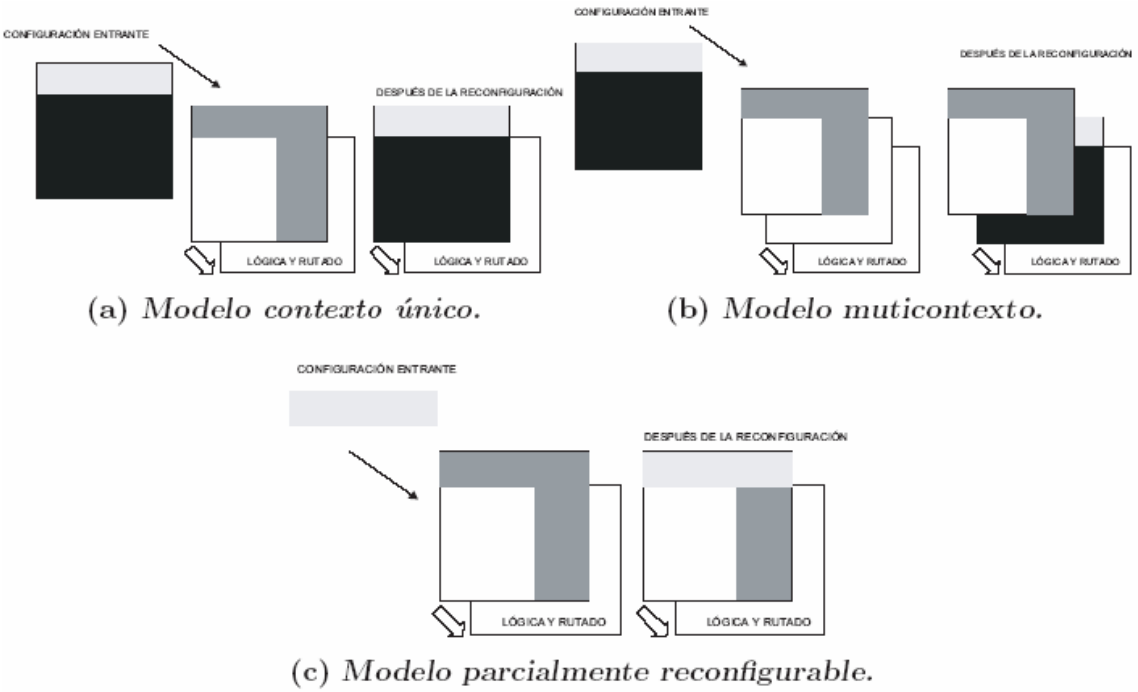
(c) *Modelo parcial y dinámico de reconfiguración.*

La figura 4(b) representa el modelo de reconfiguración dinámico. De la computación realizada por la lógica configurada (ejecución) se obtiene información que sirve para determinar la nueva configuración. En el caso de que el sistema lo permita, se podría realizar la aplicación de la nueva configuración mientras se mantiene la ejecución. En la práctica, la solución más habitual es la de mantener deshabilitada la sección que se va a reconfigurar mientras continúa la ejecución en la otra sección del dispositivo. Este modo de reconfiguración parcial dinámico se representa en la figura 4(c).

Por tanto, la reconfiguración dinámica tiene diversas variantes según el modo en el que

ésta se aplique. Las tres más representativas son:

Figura 5. Modelos de reconfiguración dinámica



(a) *Modelo contexto único.* (b) *Modelo muticontexto.*

(c) *Modelo parcialmente reconfigurable.*

- **Reconfiguración contexto único**: Corresponde con el modo de configuración de las FPGAs que únicamente soportan un acceso secuencial a la memoria de configuración. En el caso de realizar una reconfiguración dinámica con estos dispositivos se sufren unas penalizaciones temporales importantes debido a que cada intercambio de funcionalidad requiere una reprogramación completa de los mismos. El modelo de este modo se ha representado en las figuras 4(b) y 5(a). La configuración entrante sustituye completamente a la que estaba aplicada sobre la lógica configurable.

- **Reconfiguración multi-contexto**: Los dispositivos que soportan este tipo de reconfiguración tienen varios bits de memoria de configuración para cada bit de los elementos configurables [49, 50]. En la figura 5(b) se representa un modelo de estos dispositivos donde los bits de memoria pueden considerarse como múltiples planos de información de configuración. Cada plano debe configurarse totalmente, de igual forma que los de contexto único. Sin embargo, el cambio entre contextos se realiza de forma muy rápida, admitiéndose además la carga de una nueva configuración en un plano no activo mientras otro lo está.

- **Reconfiguración parcial**: Uno de los avances tecnológicos más importantes en el área de la reconfiguración consiste en la capacidad de algunos dispositivos para admitir la modificación de parte de la configuración mientras el resto del hardware sigue realizando la computación de forma ininterrumpida.

  La figuras 4(c) y 5(c) muestran este modelo de reconfiguración. En este caso el plano de configuración funciona como una memoria RAM. De este modo se pueden emplear las direcciones para especificar una determinada localización que de desea reconfigurar. La reconfiguración parcial dinámica también permite que se carguen configuraciones diferentes en áreas no usadas del dispositivo con el fin de reducir la latencia en el cambio de contexto, tal y como se propone en el trabajo de K. Danne [51].

  Existen varias plataformas reconfigurables que soportan la reconfiguración parcial como Chimaera [43], PipeRench [52], NAPA [44], Xilinx 6200 [53] y Virtex [54]. Una

variante de la reconfiguración parcial realizada con el fin de reducir la penalización por el tiempo necesario para la carga de los bitstreams parciales es la reconfiguración pipeline. En este caso, la reconfiguración ocurre en incrementos de etapas de pipeline [55]. Este sistema está orientado a computaciones de estilo datapath, donde se emplean más etapas pipeline que las que caben simultáneamente en el hardware. El caso más avanzado sería una situación donde comenzaría la computación de cada etapa tras el instante de ser programada. En este caso la configuración de cada etapa se situaría un paso a la cabeza del flujo de datos [25].

Para aliviar la problemática de la penalización en tiempo impuesta por el proceso de reconfiguración, se han investigado diversas tácticas. La precarga de configuraciones [56] es una de ellas. El objetivo en este caso es cargar la configuración en el dispositivo con anticipación a que se requiera ésta. El carácter especulativo de esta técnica fija la complejidad de la misma en determinar con suficiente antelación cuál va a ser la siguiente configuración requerida. También la compresión de la configuración se ha estudiado como una de las alternativas para la reducción del tiempo de configuración [57].

Otra alternativa abordada con el mismo fin es la del uso de caché de configuraciones en el dispositivo [58]. Al retener configuraciones en el integrado, se reduce la cantidad de información de configuración transferida. Al igual que en las cachés de los procesadores de propósito general, se aplican los conceptos de localidad temporal y espacial con el fin de decidir qué configuraciones se mantienen y cuáles se eliminan cuando se produce la reconfiguración. Decisiones incorrectas pueden producir el efecto contrario al deseado, aumentando la penalización temporal producida por la reconfiguración.

Xilinx Virtex-II Pro

Xilinx es una empresa especializada en el desarrollo de dispositivos programables (FPGAs, CPLDs, etc.) desde 1985. El modelo de FPGA utilizado en este proyecto es el Virtex-II Pro, desarrollado por Xilinx en 2004. A continuación se enumeran algunas de sus características principales:

- Arquitectura lógica programable
  - Tecnología de 130nm y 9 capas de cobre
  - Desde 3000 hasta 99000 celdas lógicas
  - Velocidad de reloj superior a los 400MHz
  - Alto rendimiento y bajo consumo
- Escalabilidad. Hasta 11 formatos.
- Características avanzadas
  - Memoria distribuida y empotrada (Flash)
  - Gestión de relojes digitales
  - Reconfiguración de la FPGA total/parcial en función de la actualización de los productos del mercado
  - Tecnología XCITE que permite mejorar la integridad de la señal y reducir espacio
- Conectividad
  - Hasta 20 *serial transceivers* (RocketIO) *full-duplex* (de 622Mbps hasta 3.125Gbps)
  - Conexiones a 100MHz mediante LVDS
- Procesamiento avanzado
  - Dos procesadores empotrados IBM PowerPC 405 a 400MHz
- Herramientas de desarrollo
  - Herramientas para la programación de la FPGA

La FPGA con la que se ha trabajado es la XC2VP30-5 FG676C de familia Virtex-II PRO de Xilinx. La familia Virtex-II se caracteriza por cargar los datos de configuración en celdas internas de memoria estática. Los valores guardados en estas celdas determinan las funciones lógicas y las interconexiones implementadas en la FPGA. El hecho de guardar los valores

dentro de estas celdas permite infinitas reprogramaciones del dispositivo. Además en esta FPGA la configuración puede cambiarse en tiempo de ejecución (reconfigurción dinámica) lo que posibilita la realización de un sistema multiprocesador hardware con varias unidades reconfigurables y uno o varios procesadores.

El nombre de la FPGA, XC2VP30-5 FG676C, indica lo siguiente:

- **Tipo de dispositivo:** XC2VP30, dispositivo 30 de la familia Virtex-II PRO.
- **Grado de velocidad:** 5, estándar.
- **Tipo de embalaje:** FG, Fine Pitch BGA 27 x 27 mm, 1.0 mm ball pitch.
- **Número de pines:** 676.
- **Rango de temperatura:** C, comercial (entre 0º y 85º).

VHDL

El VHDL (*Very High Speed Integrate Circuit Hardware Description Language*), es un lenguaje de descripción y modelado, diseñado para describir la funcionalidad y la organización de sistemas *hardware* digitales, placas de circuitos, y componentes.

La finalidad del modelado es la simulación. La sintaxis amplia y flexible del lenguaje VHDL permite tanto el modelado estructural como el modelado funcional de circuitos. En el primer caso, se describe el circuito indicando los componentes y las conexiones que lo componen (lo cual requiere un conocimiento detallado del circuito). En el segundo caso, se describe el circuito indicando lo que hace y cómo funciona, es decir, describiendo su comportamiento (sin necesidad de conocer su estructura interna).

Esta segunda metodología de modelado resulta muy interesente desde el punto de vista del diseño de sistemas digitales. Más aún teniendo en cuenta que hoy en día otra de las aplicaciones del lenguaje VHDL, con una gran demanda de uso, es la síntesis automática de circuitos. En el proceso de síntesis, se parte de una especificación de entrada con un determinado nivel de abstracción, y se desciende verticalmente por los niveles de la jerarquía de diseño hasta llegar a una implementación más detallada, menos abstracta. Puesto que el VHDL fue inicialmente concebido para el modelado de sistemas digitales, su utilización en síntesis no es inmediata. Sin embargo, la sofisticación de las actuales herramientas de síntesis es tal que permiten implementar diseños especificados en un alto nivel de abstracción.

Así pues, VHDL permite diseñar, modelar y comprobar un sistema desde un alto nivel de abstracción hasta el nivel de definición estructural de puertas lógicas. Además, siguiendo ciertas guías para síntesis, permite la implementación de diseños a nivel de puertas lógicas. Al estar basado en un estándar (IEEE Std. 1076-1987) reduce errores de comunicación y problemas de compatibilidad. Finalmente, dada su característica de modularidad, permite dividir o descomponer un diseño hardware y su descripción VHDL en unidades más pequeñas.
Los componentes de un proyecto VHDL son los siguientes:

- **Entity**: Es el más básico de los bloques de construcción en un diseño. Una entidad VHDL especifica el nombre de la entidad, sus puertos, e información relacionada con ella. Todos los diseños son creados usando una o varias entidades. La entidad describe la interfaz en el modelo VHDL.

- **Architecture**: La arquitectura describe la funcionalidad esencial de la entidad y contiene los estados que modelan el funcionamiento de ésta. Una entidad puede tener varias arquitecturas.

- **Configuration**: Permite unir la instancia de un componente a la pareja entidad-arquitectura. Describe el comportamiento a utilizar para cada entidad.

- **Package**: Es una colección de los tipos de datos y subprogramas usados comúnmente en un diseño. Las librerías forman parte de los *packages*.

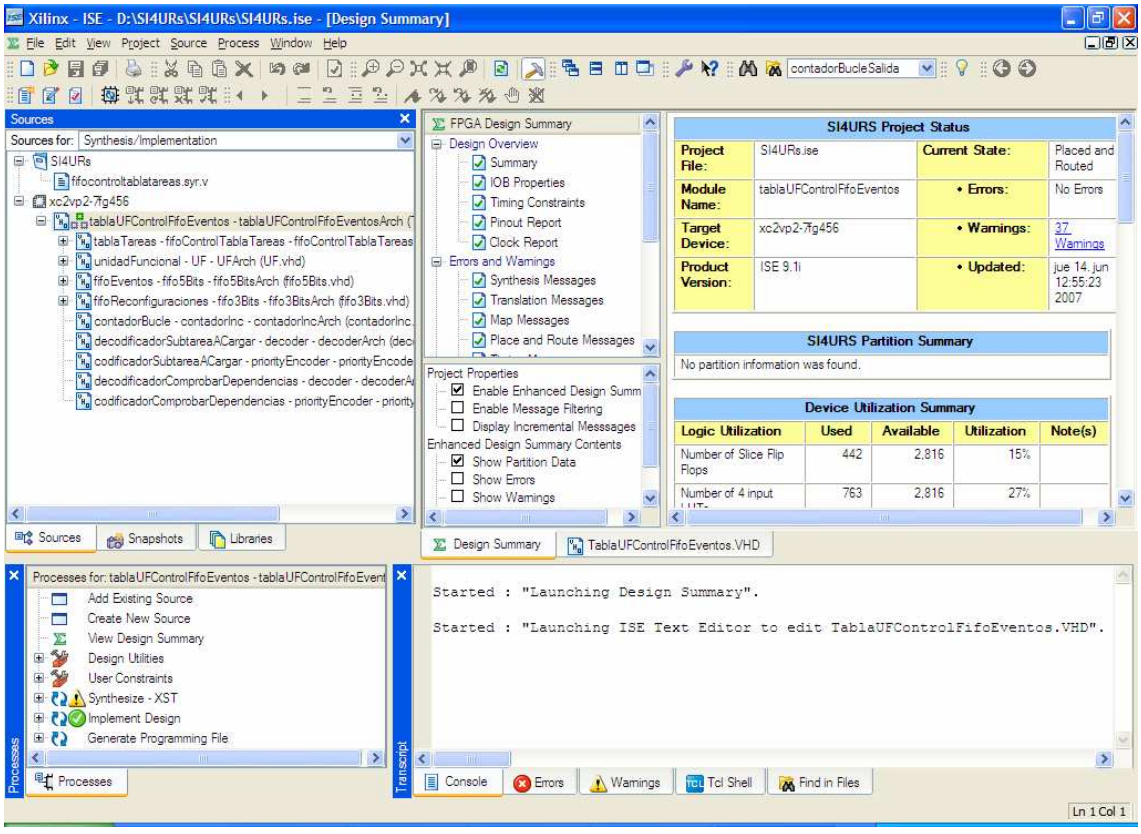
PAQUETES SOFTWARE UTILIZADOS

Para el desarrollo de la implementación en VHDL del gestor y su posterior testeo, se utilizaron los programas Xilinx ISE 9.1i y ModelSim 6.0a respectivamente. Posteriormente, se utilizó el programa Xilinx EDK 9.1i para realizar el sistema completo y un sistema SW equivalente sobre PowerPC y Microblaze.

Xilinx ISE 9.1i

El entorno de desarrollo ISE de Xilinx posee un aspecto similar al de los entornos de programación actuales como puede ser Visual Basic o Visual C, es decir, posee diversas ventanas para la visualización de tareas específicas sobre cada una de ellas. En este caso existen cuatro tipos de ventanas (figura 6):

Figura 6. Pantalla principal del entorno Xilinx ISE



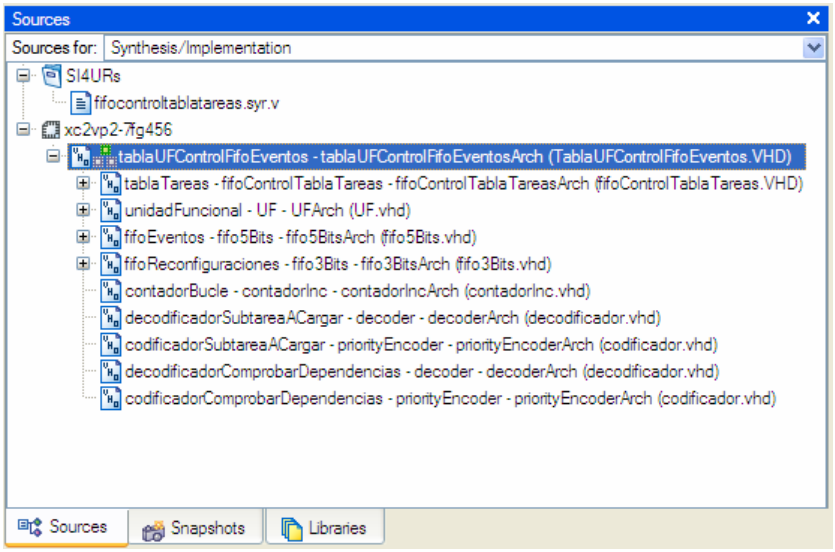
1. Ventana de ficheros fuente. En esta ventana se muestran los ficheros fuentes utilizados en el diseño y las dependencias entre ellos. También es aquí donde se elige el tipo de dispositivo donde se desea almacenar el diseño. Esta ventana posee diversas solapas para visualizar diferentes tipos de información relativa a las fuentes de diseño empleadas.
2. Ventana de Procesos. Esta ventana muestra todos los procesos necesarios para la ejecución de cada etapa de diseño. La lista de procesos se modifica dinámicamente dependiendo del tipo de fuente seleccionado en la ventana de ficheros fuente.
3. Ventanas de edición. Al hacer doble clic sobre un fichero fuente de la ventana de ficheros fuente se abre una ventana de edición para modificar el fichero (en caso de

lenguaje VHDL), o bien se ejecuta el programa que permite editar el diseño (en caso de diseños esquemáticos o máquinas de estado).

4. Ventana de información, situada en la parte inferior. Muestra mensajes de error, aviso o información emitidos por la ejecución de los programas de compilación, implementación, etc.

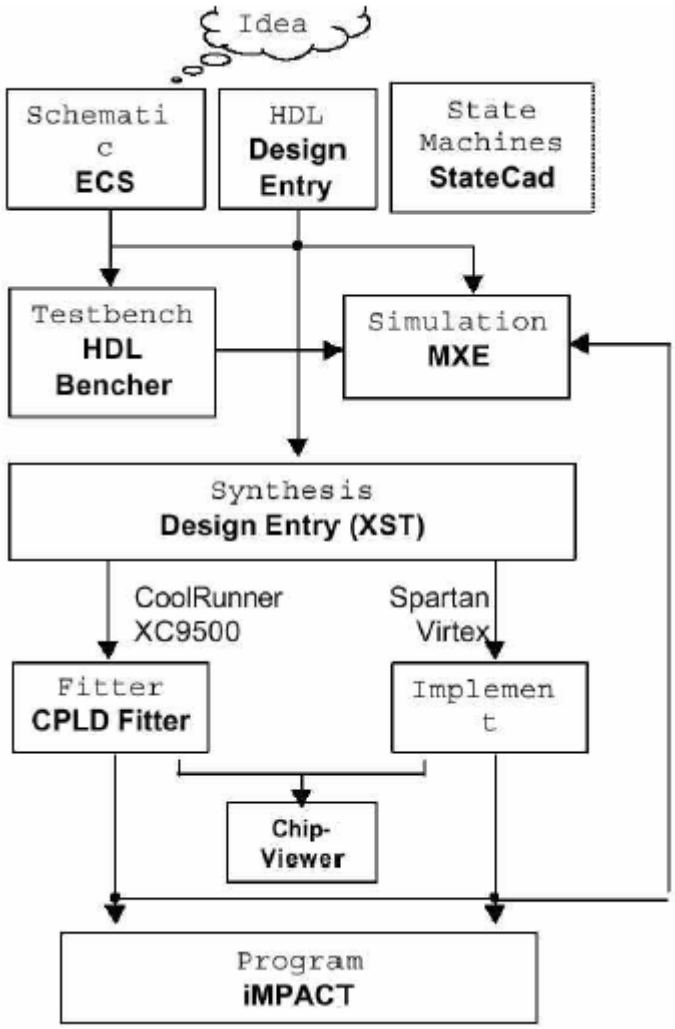
Tanto en la ventana de procesos como en la de ficheros fuente es posible modificar las opciones de cada elemento a través del botón derecho del ratón, o bien a través de los menús del entorno de diseño; estos menús se modifican dependiendo del tipo de selección realizada en las ventanas de ficheros fuente y de procesos. Cada elemento mostrado en la ventana posee un icono diferente dependiendo del tipo de acción o fichero de que se trate, por ejemplo, nos indica si un elemento es un documento de texto, una acción para ejecutar en el entorno ISE o una acción para ejecutar por un programa adicional como puede ser ModelSim a la hora de simular. También muestra información sobre el estado que ha dado resultado tras la ejecución del proceso, es decir, si ha sido satisfactorio, si ha tenido errores, o ha tenido avisos. Las imágenes de la figura 7 muestran un ejemplo de los diferentes tipos de información mostrados en estas dos ventanas. Para la ventana de ficheros fuente, se indica si un fichero es de código, de vectores de test, si es un paquete, o una selección de dispositivo.

Figura 7. Detalle de las ventanas de ficheros fuente y de procesos



En concreto, la ventana de procesos incorpora todas las opciones necesarias para realizar todos los pasos de implementación de sistemas en lógica programable, incluyendo la edición y verificación. La figura 8 muestra el diagrama de flujo de diseño en Xilinx ISE, y la figura 9 muestra las diversas partes en que se divide la ventana de procesos dependiendo de la tarea a realizar.

Figura 8. Proceso de diseño en Xilinx ISE



De manera resumida, el proceso de diseño resulta sencillo y se realiza en tres pasos, el primero consiste en añadir los ficheros fuente, en el segundo paso se selecciona el fichero de más alto nivel que se quiere implementar, y finalmente se hace doble clic sobre el último proceso al que se desea llegar, de este modo se ejecutarán todos los procesos intermedios necesarios para llegar al proceso seleccionado en último lugar (figura 10).

Figura 9. División de tareas dentro de la ventana de procesos

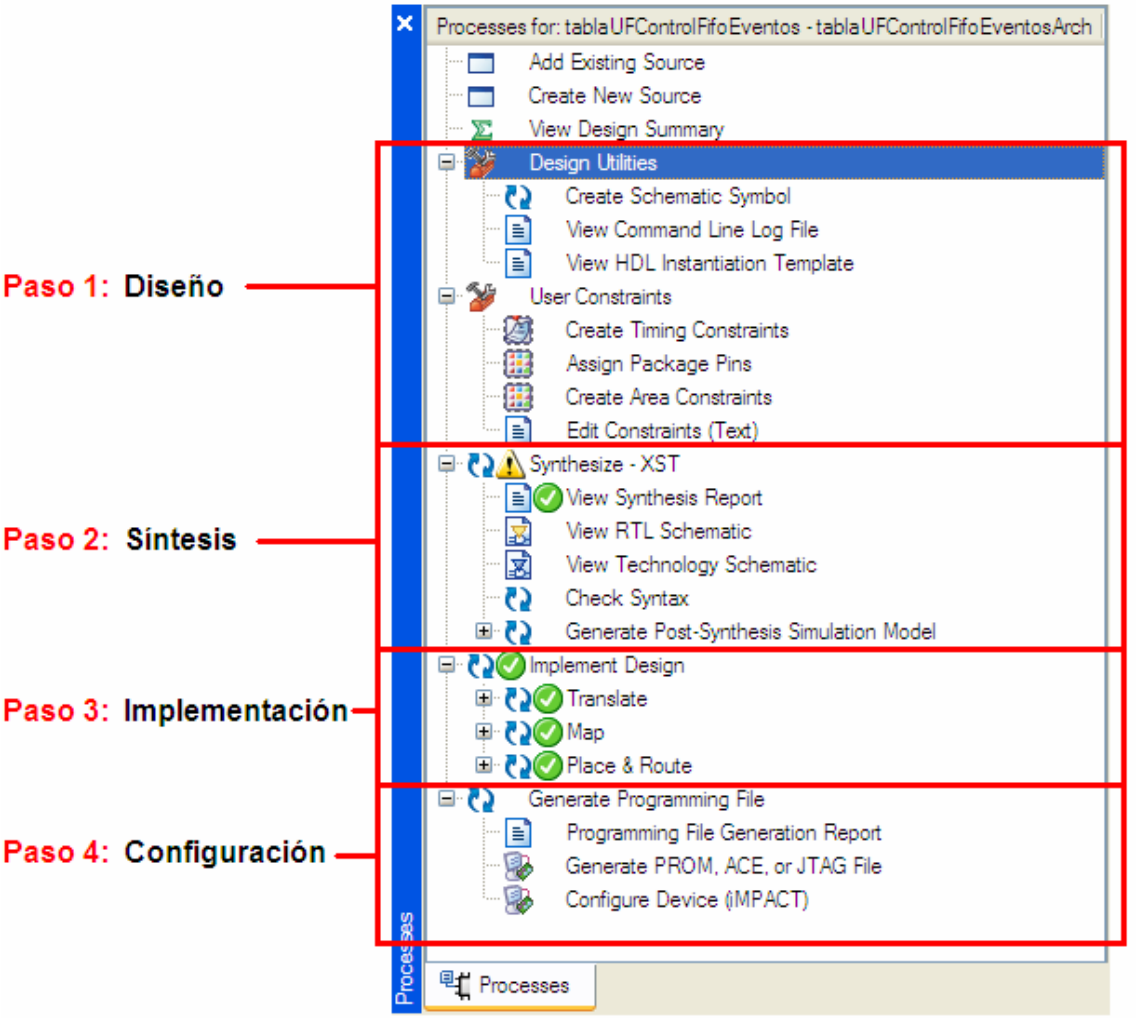
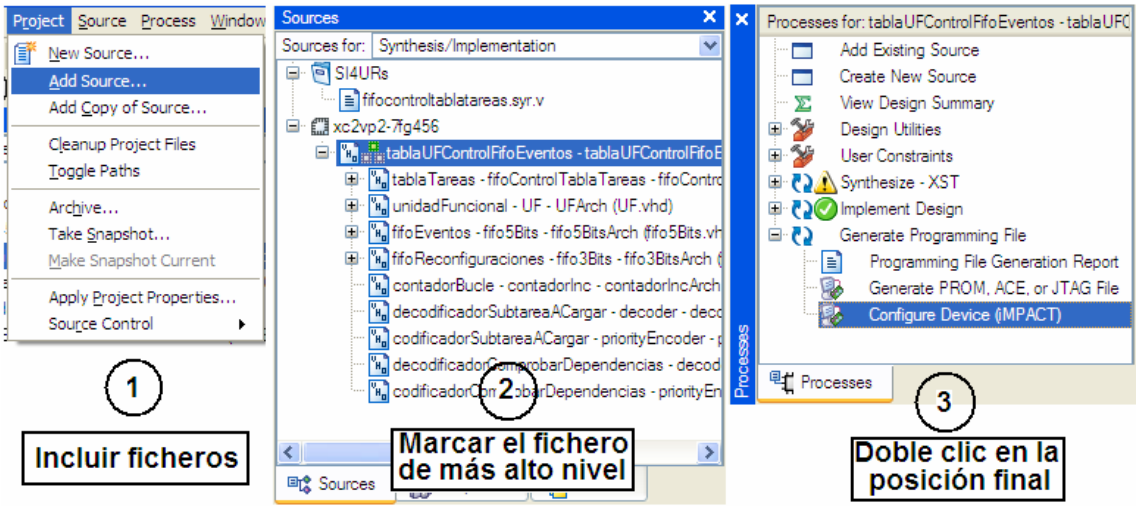


Figura 10. Proceso simplificado para el desarrollo de un diseño en Xilinx ISE



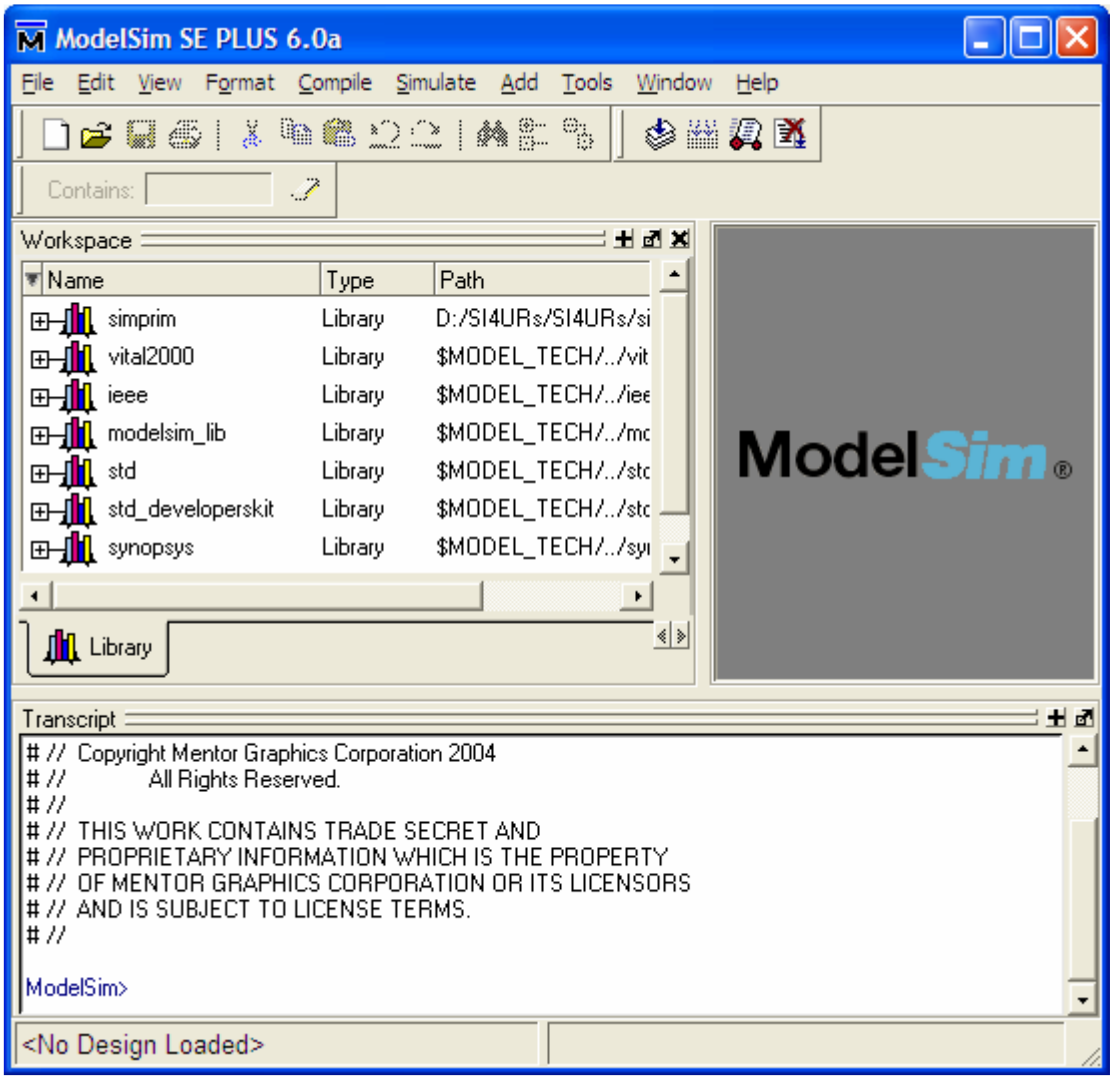
En el proyecto, este software se utilizó para la implementación en VHDL del gestor y para el testeo de los módulos básicos, con el propio simulador que incorpora ISE.

ModelSim 6.0a

Los bancos de prueba (o *testbenchs*) son una parte esencial del proceso de diseño, ya que permiten comprobar su correcto funcionamiento y ayudan en la automatización del proceso de verificación del diseño. Recoger estadísticas de la cobertura del código durante la simulación ayuda a asegurar la calidad y la minuciosidad de las pruebas.

El software ModelSim SE 6.0 utilizado en este proyecto, proporciona un entorno integrado de depuración (*Integrated Debug Environment*) que facilita el depurado eficiente de diseños basados en FPGAs, programados en cualquiera de los tres lenguajes siguientes: VHDL, Verilog y SystemC.

Figura 11. Ventana principal del simulador ModelSim SE 6.0



Además de simular el funcionamiento del sistema diseñado, el ModelSim SE 6.0 permite visualizar las señales de cada puerto del bloque simulado y realizar las modificaciones necesarias del código en función del resultado obtenido.

Partiendo del código proporcionado por el programa Xilinx ISE 9.1i, la simulación se puede realizar a distintos niveles: desde el nivel más alto en el que se utilizan modelos “ideales” de los componentes hasta el nivel Post Place & Route donde cada componente ha sido mapeado a

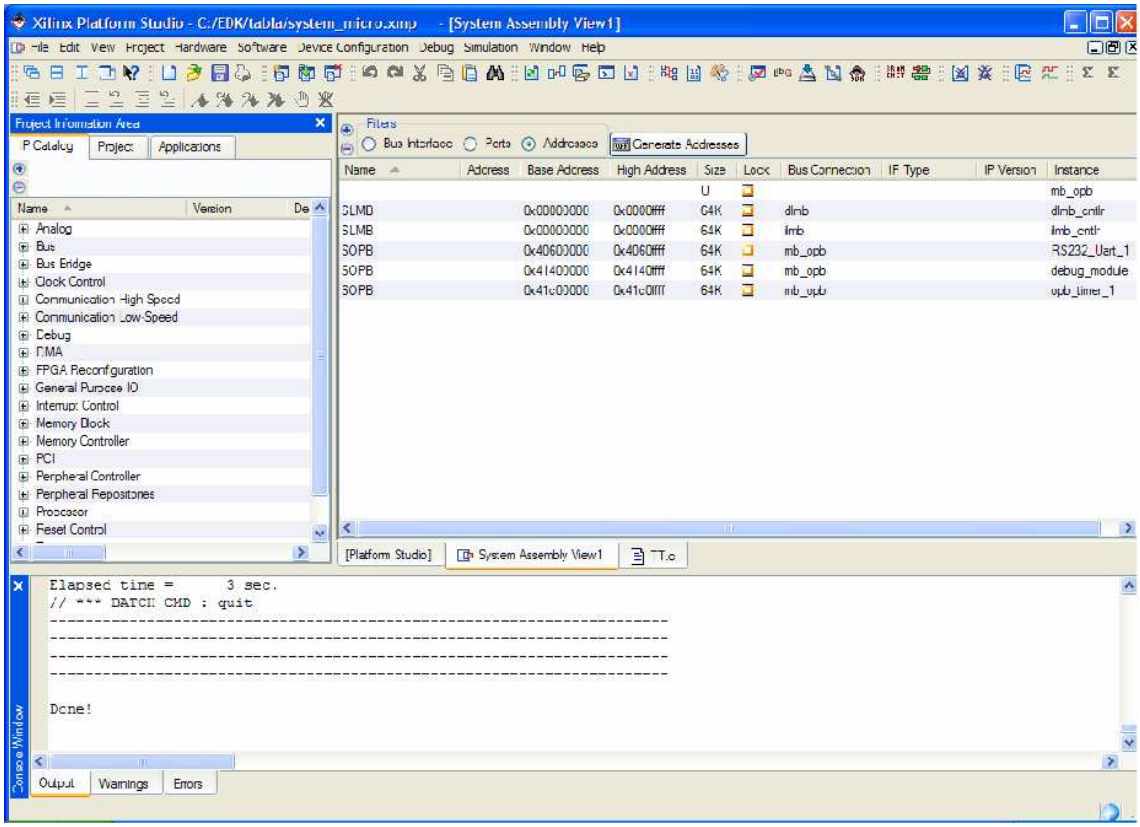
un elemento concreto de la FPGA y posteiormente han sido conectadas entre sí, de forma que la simulación se basa en modelos muy precisos de los componentes que incluso incluyen los retardos de las conexiones.

Xilinx EDK 9.1i

Xilinx EDK es la abreviatura de Xilinx Embedded Development Kit y como indica la palabra es un conjunto de herramientas para el diseño y desarrollo de sistemas embebidos. Esta herramienta de diseño se encuentra dividida en tres subbloques o subpestañas principales. En el primer de ellos, encontramos un catálogo con todos los elementos de que consta nuestro diseño "IP Catalog"; el segundo subbloque consta de la información del projecto "Project" y el tercer y último subbloque "Applications", muestra las aplicaciones (en código C o C++) asociadas a nuestro diseño.

Asimismo contamos con la barra de herramientas y de iconos para generar el fichero .bit del proyecto, guardar, cargar, descargar dicho fichero a la fpga y muchas más opciones.

Figura 12. Xilinx EDK



RECONFIGURACIÓN DINÁMICA

Opciones arquitectónicas

El HW reconfigurable ha captado la atención de un amplio número de grupos de investigación desde su aparición. Está ganando popularidad tanto a nivel académico, donde cada vez son más los grupos de investigación dedicados a su estudio, como a nivel comercial. Este considerable desarrollo ha propiciado que existan numerosas opciones arquitectónicas por las que hay que decantarse a la hora de realizar un diseño HW que convendrá comentar brevemente. En particular, podemos optar por los siguientes criterios:

1) ***Grano fino o grano grueso.*** Las arquitecturas de grano fino permiten cambiar la configuración para alterar el comportamiento de la placa a nivel de bit. Por otro lado, las arquitecturas de grano grueso trabajan con palabras de varios bits; partiendo del hecho de que las operaciones se realizan a nivel de palabra. El grano fino proporciona máxima flexibilidad para crear una configuración óptima de un algoritmo, a cambio de pagar un precio en área, longitud y cantidad de conexiones existentes, tiempo de reconfiguración, consumo de energía e incluso el rendimiento. El grano grueso representa el extremo opuesto.

2) ***Utilización de uno o múltiples contextos***. Una FPGA de un contexto es aquélla que sólo guarda la información de una reconfiguración en un instante de tiempo dado. Por otro lado, la utilización de múltiples contextos implica que la plataforma dispone de la memoria suficiente como para guardar múltiples configuraciones, así como la capacidad de cambiar de uno a otro en tiempo de ejecución cada vez que sea necesario. La inmensa mayoría de las FPGAs del mercado sólo soportan un único contexto, debido al sobrecoste de área de almacenamiento de los diferentes contextos, así como la lógica necesaria para la selección de los mismos.

3) ***Reconfiguración parcial o global.*** Hablamos de reconfiguración global cuando las reconfiguraciones de la FPGA se han de realizar simultáneamente en toda la placa. Por otro lado, cuando se permite que una parte de la plataforma pueda cambiar su funcionalidad mientras el resto permanece inalterado se habla de reconfiguración parcial. Esta última tiene dos ventajas importantes: el tiempo de reconfiguración es menor (al ser sólo parte del HW de la placa el involucrado) y permite gestionar de forma independiente la ejecución de múltiples tareas.

Para el desarrollo de este proyecto se ha optado por un modelo de grano fino combinado con algunos componentes empotrados dentro de la FPGA, al representar un compromiso entre consumo de recursos y rendimiento. Utilizamos un único contexto, ya que las FPGAs de que disponíamos sólo operaban de ese modo. Finalmente, hacemos uso de reconfiguración parcial, elección clave para que se pueda desarrollar un sistema que gestione la reconfiguración y la ejecución de tareas en varias unidades reconfigurables y/o varios procesadores.

Hacia un sistema con varias unidades reconfigurables

En la planificación típica de un sistema operativo, la existencia de muchos procesos y/o *threads* puede llegar a degradar considerablemente el rendimiento del sistema. Como posibles soluciones a este problema, se pueden plantear, fundamentalmente, dos mejoras: aumentar el número de procesadores y migrar tareas SW a HW. Ambas propuestas buscan paralelizar a nivel de datos y a nivel de cómputo.

El HW dinámicamente reconfigurable permite llevar a cabo estas dos mejoras (ejecución SW/HW), cuyas ventajas e inconvenientes son:

Ventajas: Se realiza una ejecución totalmente en paralelo y permite combinar de manera razonable una ejecución rápida con consumo de energía aceptable.

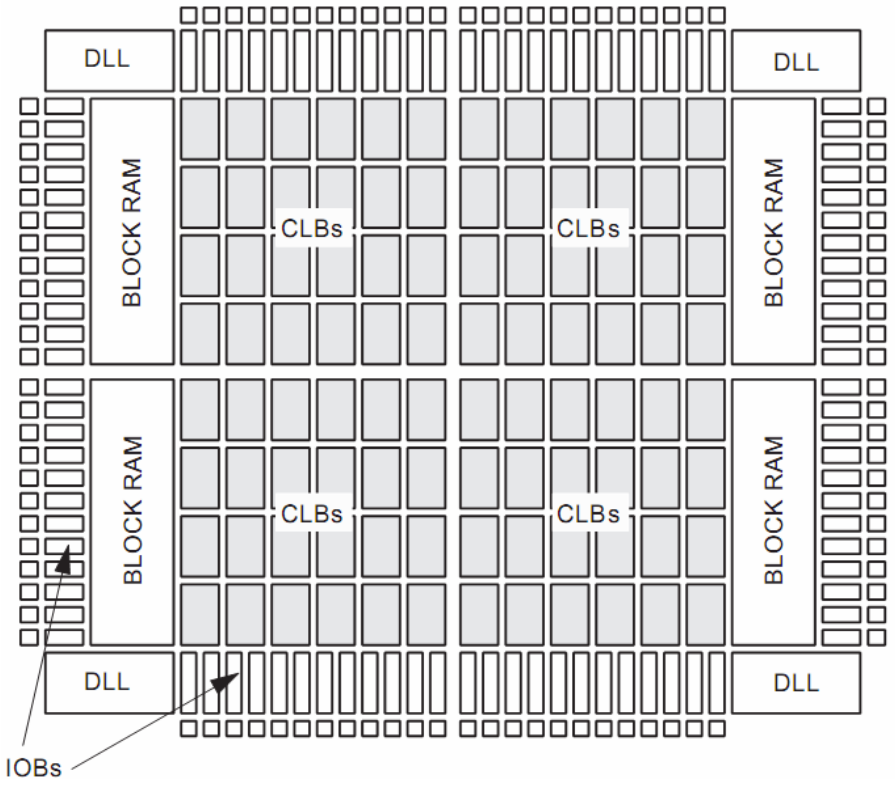
Inconvenientes: No es trivial pasar de SW a HW, así como tampoco lo es reconfigurar parcialmente y en tiempo de ejecución parte de la FPGA. Asimismo, las latencias de reconfiguración pueden degradar el rendimiento del sistema. Esto exite una buena planificación que las oculte; y estas planificaciones suelen ser costosas: A menudo hay que encontrar una planificación casi óptima para que merezca la pena llevarla a cabo y se manejan estructuras de datos complejas, como listas, grafos…

Quizá las características más interesantes del HW dinámicamente reconfigurable sean su versatilidad y el hecho de que se puedan ejecutar tareas totalmente en paralelo manteniendo un compromiso entre tiempo de reconfiguración y/o ejecución y consumo de energía. Unido a que queda aún por realizar en este campo un amplio trabajo de investigación (la gran parte de las posibilidades que ofrece el HW dinámicamente reconfigurable están todavía desaprovechadas debido a la falta de soporte que proporcionan los fabricantes) nos ha parecido suficiente como para habernos decantado por la realización de una labor de investigación propia en este campo desarrollando un gestor que permita la ejecución de tareas en una plataforma en HW dinámicamente reconfigurable siguiendo una planificación dada.

Reconfiguración parcial dinámica en dispositivos Virtex

En los últimos años, la familia de dispositivos FPGA Virtex ha sido la alternativa más utilizada para experimentación con lógica dinámica y parcialmente reconfigurable basada en FPGAs comerciales de grano fino.

Figura 13. Distribución de los recursos lógicos en el dispositivo más pequeño de la familia Spartan-II

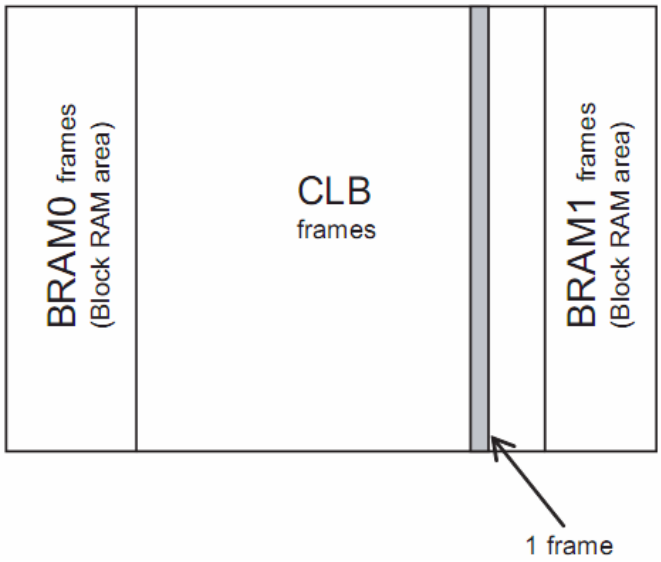


La estructura interna de estas FPGAs ha sido descrita en un apartado anterior de esta memoria. En la figura 13 se representa la distribución de los recursos lógicos programables en un dispositivo de la familia Spartan-II. Se diferencian cuatro tipos de recursos: los bloques lógicos configurables (*Configurable Logic Blocks* -CLBs-) cuyos elementos están divididos en dos *slices*; los bloques de entrada-salida (*Input/Output Blocks* -IOBs-); los bloques de memoria RAM dedicados (BRAMs); y los dispositivos para la gestión del reloj. Éstos, en concreto, para la familia Spartan-II son los bucles enganchados en retardo (*Delay-Locked Loops* -DLLs-) y para la Virtex los gestores digitales de reloj (*Digital Clock Managers* -DCMs-).

Los recursos lógicos en la FPGA están distribuidos de forma regular formando una matriz cuyas posiciones se identifican en la nomenclatura de Xilinx con una “R” indicando la fila (*row*) y una “C” para identificar a la columna (*column*).

La configuración de estos elementos y del rutado se escribe en la memoria de configuración SRAM de la FPGA. La mínima sección del dispositivo que se puede cambiar en una única operación de reconfiguración parcial está definida por un *frame*. Este término, propio de Xilinx, especifica una unidad de la memoria SRAM de configuración de la FPGA. Un *frame* tiene un bit de anchura y se extiende desde la fila superior de la matriz hasta la fila inferior.

Figura 14. Distribución de los diferentes tipos de frames en un dispositivo Virtex



La figura 14 muestra la distribución de dos tipos de *frames* en la memoria de configuración de la FPGA. Se distinguen tres zonas: a la izquierda y a la derecha, BRAM0 y BRAM1 incluyen los *frames* para la configuración de los datos de los bloques de RAM dedicados (BRAM *frames*) y en el centro de la matriz se sitúan los *frames* de configuración de los CLBs (CLB *frames*).

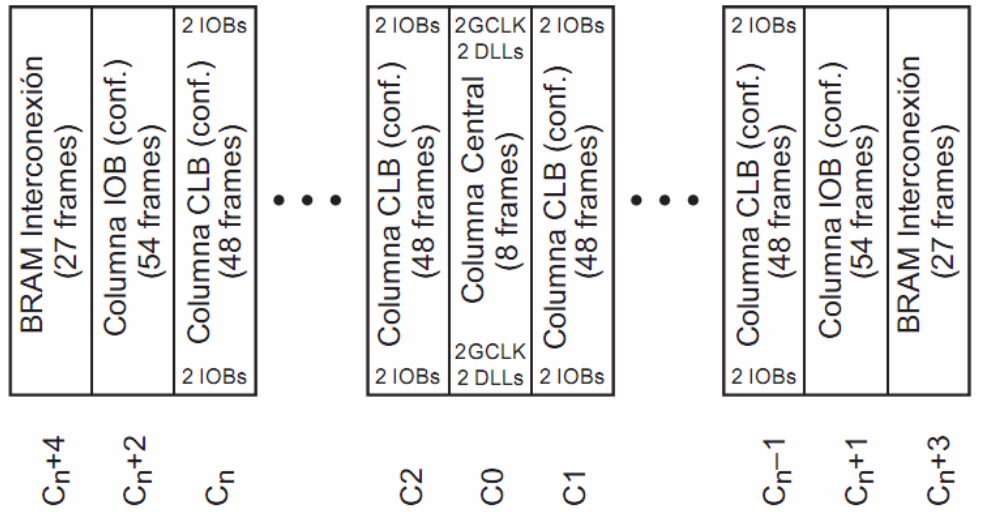
Los CLB *frames* contienen todos los datos de configuración para todos los elementos programables de la FPGA incluyendo los valores de las LUTs, lógica auxiliar, IOBs, elementos de control de los BRAMs y las interconexiones. De esta forma, se puede acceder a la configuración de estos recursos mediante una operación de escritura sobre el puerto de configuración de la FPGA, sin interrupción en el funcionamiento del circuito.

Además, como el contenido inicial de los biestables de los CLBs no se especifica en el *bitstream* de configuración, y por tanto, tampoco en el *frame*, los datos almacenados en los mismos no se ven modificados en el proceso de reconfiguración parcial.

En cuanto a los bloques de memoria RAM dedicada, se debe tener en cuenta que al escribir los BRAM *frames* en la memoria de configuración se está alterando su contenido. Por ello, si la aplicación accede a los mismos durante este periodo transitorio puede leer valores no

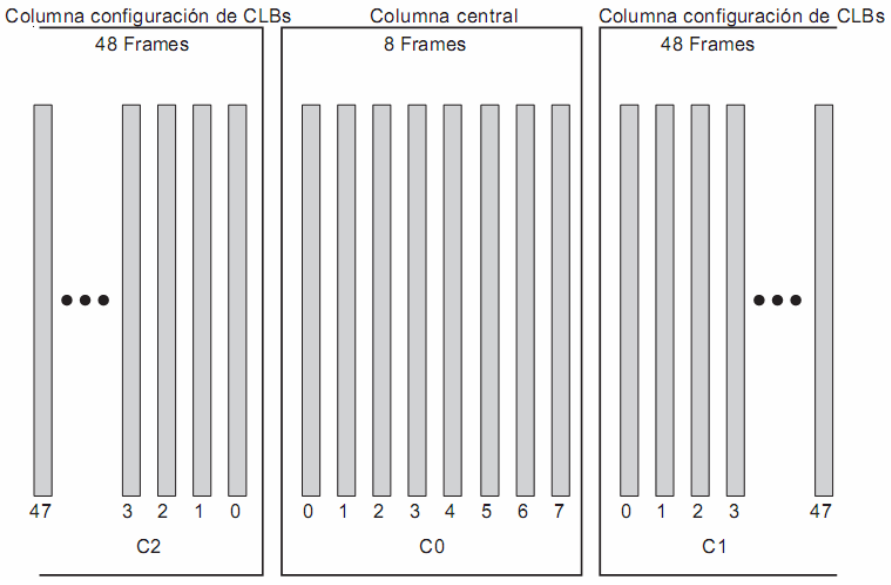
válidos. Esta situación debe ser tenida en cuenta por el sistema de control de la reconfiguración que se esté utilizando en el circuito.

Figura 15. Distribución de las *frame columns* en la memoria de configuración



Los *frames* están agrupados en unidades superiores denominadas *frame columns* [59, 60]. El área de la memoria de configuración de los CLB frames tiene cuatro categorías de *frame columns*: una columna central de 8 *frames*, columnas de 48 *frames* para la configuración de los CLBs, dos columnas de 27 *frames* que especifican el conexionado de los BRAMs y dos columnas para la configuración de los IOBs (54 *frames* por columna). En la figura 15 se muestra la distribución y numeración de las *frame columns* en la memoria de configuración, siguiendo una distribución de *ping-pong*.

Figura 16. Distribución de los *frames* en los *frame columns*



Dentro de cada *frame column*, los *frames* se numeran secuencialmente desde el frame que se encuentre más cercano al centro hasta el más alejado, tal y como se muestra en la figura 16.

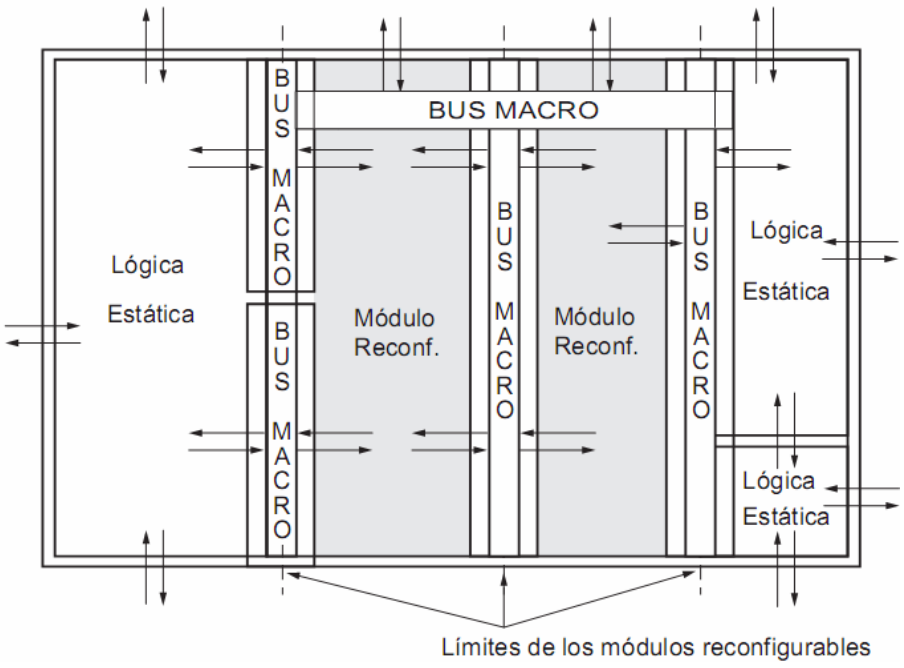
La dirección de un *frame* en la memoria de configuración está compuesta por dos números: el *major address*, que es el número de *column frame*; y el *minor address*, que es el número del *frame* en esa columna. Por tanto, los *bitstream* parciales de configuración incluyen las

direcciones y los contenidos de los frames que se deben modificar, además de comandos específicos para el puerto de configuración.

Pero para poder aplicar la reconfiguración parcial dinámica sobre estos dispositivos, se deben tener en cuenta las siguientes restricciones generales [61]:

- La altura de una sección reconfigurable deberá ser la de toda la matriz de recursos lógicos.
- La anchura ocupada por un módulo reconfigurable deber ser múltiplo de 4 slices (2 CLBs). Por ejemplo, un módulo reconfigurable podría comenzar en coordenadas X=0, 4, 8, 10, etc. con anchuras de w=4, 8, 12, etc. slices.
- Toda la lógica que se defina en la anchura ocupada por ese módulo reconfigurable pertenece a ese módulo. Esto incluye tanto a los elementos que forman parte del CLB como a los *buffers* triestado (*Tbufs*), IOBs, multiplicadores y rutado.
- La lógica de reloj (*buffers globales*, DCM o DLL, etc.) tiene *frames* separados de los utilizados para configurar los CLBs y BRAMs.
- La configuración de los IOBs situados en el lateral derecho y en el izquierdo de la matriz reconfigurable se realiza mediante los CLB *frames* correspondientes a las columnas extremas.
- Los límites definidos para un determinado módulo reconfigurable no pueden ser cambiados dinámicamente, siendo fija la posición y el área que ocupa.
- La comunicación entre distintos módulos reconfigurables se debe realizar utilizando unos elementos fijos que definan rutas únicas de configuración. Estas herramientas se denominan *Bus-Macros* y se encuentran pre-rutadas antes de realizar la fase de emplazamiento y rutado de todo el diseño de la FPGA (figura 17).

Figura 17. Conexión entre módulos reconfigurables para aplicar reconfiguración *inter-task* en dispositivos Virtex de Xilinx



Esta restricción se debe tener en cuenta cuando el tipo de reconfiguración parcial dinámica que se pretende aplicar es de tipo *inter-task*, lo que obliga al cambio completo del módulo reconfigurable.

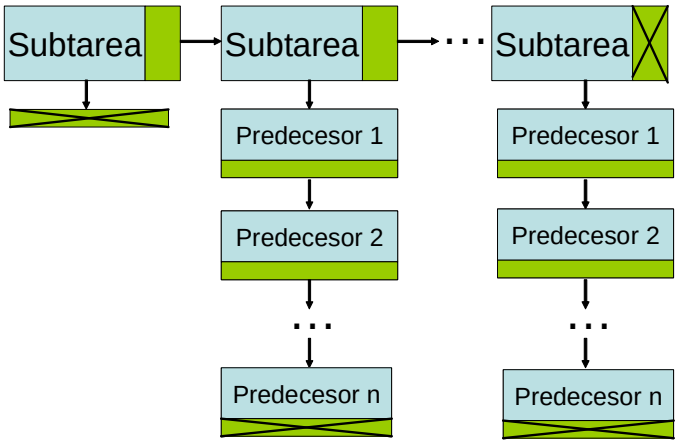
- Las señales globales de *reset* y *set* no pueden utilizarse de manera local sobre un módulo reconfigurable, lo que implica que sea necesario disponer de señales de inicialización independiente para cada módulo.

EJEMPLO DE MOTIVACIÓN

En co-diseño HW/SW con frecuencia las tareas se representan como grafos de subtareas, donde una subtarea es la unidad básica de planificación, que representa un núcleo con carga computacional significativa (uno o varios bucles importantes) que pueden ser asignado a una unidad reconfigurable, y las aristas del grafo representan las dependencias entre subtareas. Para controlar la ejecución de estas tareas, el procesador debe manejar estructuras de datos complejas en tiempo de ejecución, con una gran carga computacional. Por ejemplo, la figura 18 muestra la representación de un grafo de subtareas usando una lista enlazada. En este caso la lista principal incluye todas las subtareas del grafo, y cada nodo contiene otra lista con todos sus predecesores. Si el planificador tiene que comprobar si una tarea dada está lista para su ejecución, debe buscar primero el nodo correspondiente en la lista, y luego leer el número de predecesores. Después de esto, debe recorrer la lista de predecesores entera, y mirar el estado de todos ellos para comprobar si ya terminaron su ejecución. La complejidad de esta operación es de $\Theta(N^2)$, donde N es el número de subtareas en el grafo.

Por supuesto, hay muchas maneras de mejorar la eficacia de esta operación, como la utilización de tablas indexadas para permitir un acceso directo a cada nodo del grafo en lugar de seguir los punteros, o actualizar la lista de predecesores cada vez que una subtarea termine su ejecución. Sin embargo, el manejo de datos complejos será necesario en cualquier caso.

Figura 18. Grafo de subtareas utilizando una lista enlazada



Además, dado que el SO y el planificador deben interactuar con las URs, frecuentemente se realizarán comunicaciones HW/SW. Esta comunicación implicará probablemente manejo de interrupciones, y utilizar un bus compartido que con frecuencia es el cuello de botella del sistema.

El gestor desarrollado proporciona dos ventajas principales para reducir este problema. En primer lugar, el gestor controla directamente la ejecución de un grafo de subtareas en las URs, y gracias a su tabla asociativa operaciones como la actualización del grafo de subtareas, o la comprobación de si una subtarea está lista para ejecución se realizan rápidamente: un solo ciclo de reloj para actualizar cada dependencia y un solo ciclo de reloj para comprobar si una

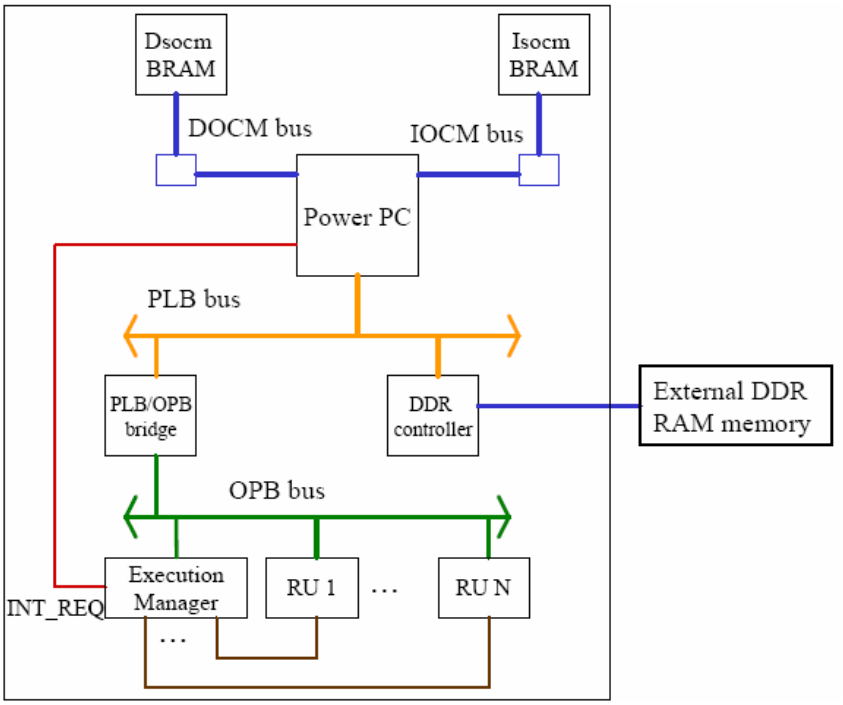
subtarea está lista. Segundo, el gestor propuesto sólo tiene que comunicarse con el procesador dos veces durante la ejecución de un grafo, una vez al principio de la ejecución, para recibir toda la información, y otra vez al final, para informar al procesador que la ejecución se realizó correctamente.

DESARROLLO DEL PROYECTO

VISIÓN GENERAL DEL SISTEMA

La implementación se ha desarrollado considerando el gestor como un componente HW incluido como un periférico on-chip conectado con el procesador a través de un bus y con las URs a través de comunicaciones punto a punto. Se ha implementado utilizando el entorno de desarrollo Xilinx EDK 9.1i porque proporciona soporte para desarrollar e integrar periféricos; así como compilar, simular y depurar proyectos SW escritos en C o en C++. Además, facilita el manejo de interrupciones, comunicaciones HW/SW (usando la interfaz del periférico y la jerarquía proporcionada por el bus), transacciones por DMA y medidas de tiempos de ejecución (ciclos).

Figura 19. Esquema de la arquitectura con del gestor HW



La figura 19 muestra una visión general del sistema completo con un gestor HW incluido como un periférico en el sistema. El Power PC es un procesador empotrado en la FPGA y el gestor y las URs se incluyen como periféricos en el sistema conectados a traves de la línea de interrupciones y la jerarquía de buses. En este caso las URs no se comunican directamente con el procesador, sino que interactúan directamente con el gestor de ejecución. Este esquema reduce drásticamente el número de interrupciones que el procesador debe gestionar; ya que cuando la plataforma ejecuta un nuevo grafo de tareas, el procesador envía la información al gestor incluyendo la descripción del grafo de tareas y la planificación seleccionada. Con esta información el gestor guía la ejecución del grafo, teniendo en cuenta los eventos generados por las URs, y finalmente genera una interrupción para informar al procesador. El entorno EDK soporta dos opciones fáciles de utilizar para enviar datos al gestor: una FIFO incluida en la interfaz del gestor y una transacción por DMA. La segunda opción es óptima en prestaciones, ya que el procesador no tiene que escribir directamente la información en la FIFO, sino que sólo necesita enviar la dirección inicial y el tamaño de los datos. Por otro lado, el controlador DMA genera una penalización en términos de área (tabla 6); sin embargo, de acuerdo con los experimentos realizados, una transacción por DMA es casi 3.5 veces más rápida que una transacción de datos equivalente entre el procesador y la FIFO local al periférico.

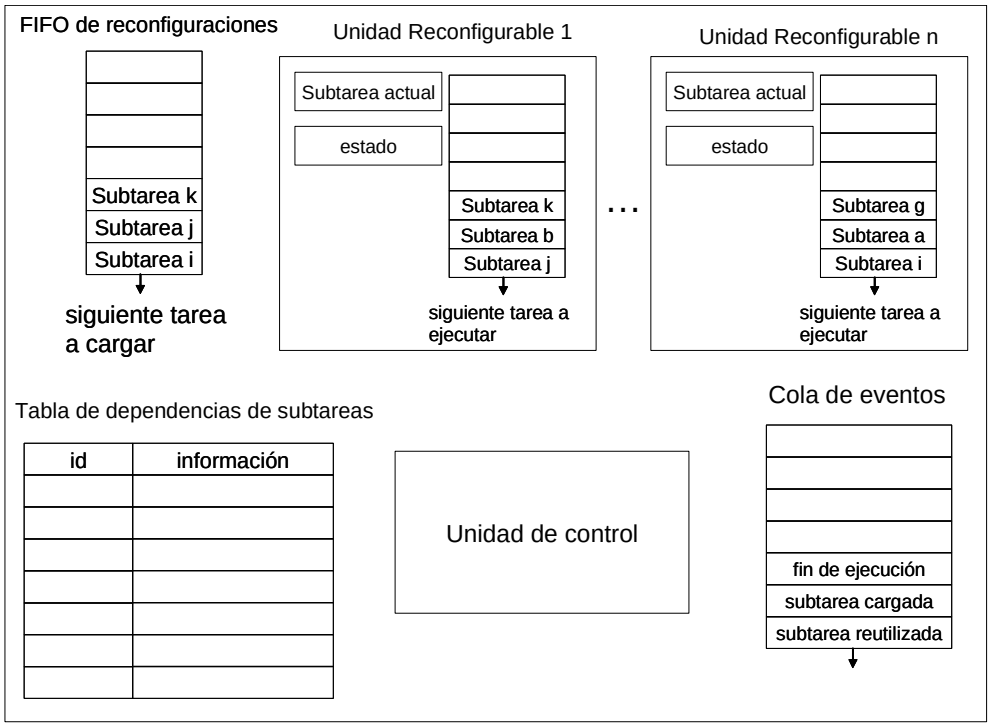
Dado que no existen en la actualidad sistemas multitarea HW hemos desarrollado una plataforma que simula estos sistemas utilizando contadores programables. Estos contadores se

incluyen en el sistema como periféricos y se utilizan para simular las latencias de reconfiguración y el tiempo de ejecución de cada subtarea. Cada UR incluye dos contadores (uno para las reconfiguraciones y otro para la ejecución) Cuando un contador termina su cuenta avisa al sistema de que ha finalizado. De esta forma se generan los eventos de fin de ejecución y de fin de reconfiguración en tiempo de ejecución.

VISIÓN GENERAL DEL GESTOR

El sistema es un gestor de ejecución de tareas de un procesador basado en HW reconfigurable. En la figura 20 se puede observar la organización del gestor y seguidamente se procederá a la descripción de cada una de sus partes.

Figura 20. Esquema del gestor



Sus componentes son:

- **Tabla de dependencias entre subtareas:** Cuando se debe ejecutar una nueva tarea (representada mediante un grafo) las dependencias de cada subtarea se almacenan en una tabla asociativa. Cada vez que una subtarea finaliza su ejecución, todos sus sucesores se actualizan, y esa tarea se elimina de la tabla. Esta tabla se consulta antes de comenzar a ejecutar una subtarea para comprobar si todas las dependencias están resueltas, es decir, si las subtareas predecesoras se han ejecutado ya. Esta operación se puede realizar en un solo ciclo de reloj.

- **Información de la unidad:** El gestor asigna una FIFO y dos registros a cada unidad reconfigurable. La FIFO almacena las subtareas asignadas a esa unidad siguiendo la planificación dada, mientras que los dos registros almacenan el estado de la unidad y la subtarea que en ese momento allí esté cargada, respectivamente.

- **FIFO de reconfiguraciones:** Esta FIFO almacena la secuencia de subtareas que deben ser reconfiguradas.

- **Cola de eventos:** Almacena los eventos que se generan en tiempo de ejecución, como el fin de la ejecución de una subtarea o el fin de una reconfiguración.
- **Unidad de control:** Esta unidad lee los eventos para identificar si una subtarea puede comenzar su ejecución o si es posible anticipar una configuración.

El sistema es escalable en cuanto a número de unidades reconfigurables, número máximo de predecesores y/o sucesores que puede tener una subtarea y longitud de los identificadores de las subtareas. Esto ha permitido hacer pruebas con diferentes valores para estos parámetros y así determinar cuáles de estos valores son los más adecuados para un funcionamiento óptimo.

DESCRIPCIÓN DEL GESTOR

FIFOs

1) Las BLOCK RAMs

En primer lugar, mencionar que, al ser las FIFOs dispositivos de almacenamiento masivo de datos, es muy probable que si no se hace un uso eficiente de los recursos disponibles en la FPGA, el diseño ocupará demasiada área de integración y será poco eficiente.

Para subsanar este problema se ha utilizado un componente de memoria de Xilinx llamado *RAMB16Sn*, en donde n puede ser: 1, 2, 4, 9, 18 y 36. No todas las FPGAs los tienen; sin embargo, sólo hay que consultar la tabla 1 para asegurarnos de que en una Virtex XC2VP30-5 FG676C, que es la placa que se utilizará, este componente está disponible:

Tabla 1. Disponibilidad del módulo RAMB16Sn en distintos modelos de FPGA

Spartan-II, Spartan-IIE	Spartan-3	Virtex, Virtex-E	Virtex-II, Virtex-II Pro, Virtex-II Pro X	XC9500/XV/XL	CoolRunner XPLA3	CoolRunner-II
N/A	Primitive	N/A	Primitive	N/A	N/A	N/A

Los componentes *RAMB16_S1, RAMB16_S2, RAMB16_S4, RAMB16_S9, RAMB16_S18* y *RAMB16_S36* son bloques de memoria de acceso aleatorio con escritura asíncrona. Tienen 16384 bits de memoria para datos. Por su parte, el *RAMB16_S9, RAMB16_S18* y el *RAMB16_S36* tienen además 2048 bits adicionales de paridad que no utilizaremos. Las posibles configuraciones del *RAMB16_Sn* son las que se pueden observar en la tabla 2, mientras que su comportamiento es el que se puede observar en la tabla 3.

Tabla 2. Posibles configuraciones del módulo RAMB16Sn

Componente	Celdas de datos		Celdas de paridad		Bus de direcciones	Bus de datos	Bus de paridad
	Profundidad	Anchura	Profundidad	Anchura			
RAMB16_S1	16384	1	-	-	(13:0)	(0:0)	-
RAMB16_S2	8192	2	-	-	(12:0)	(1:0)	-
RAMB16_S4	4096	4	-	-	(11:0)	(3:0)	-
RAMB16_S9	2048	8	2048	1	(10:0)	(7:0)	(0:0)
RAMB16_S18	1024	16	1024	2	(9:0)	(15:0)	(1:0)
RAMB16_S36	512	32	512	4	(8:0)	(31:0)	(3:0)

Tabla 3. Comportamiento del módulo RAMB16Sn

Entradas								Salidas			
GSR	**EN**	**SSR**	**WE**	**CLK**	**ADDR**	**DI**	**DIP**	**DO**	**DOP**	**Contenido de la RAM**	
										Datos en la RAM	**Paridad en la RAM**
1	X	X	X	X	X	X	X	INIT	INIT	S/C	S/C
0	0	X	X	X	X	X	X	S/C	S/C	S/C	S/C
0	1	1	0	↑	X	X	X	SRVAL	SRVAL	S/C	S/C
0	1	1	1	↑	dir	datos	Pdatos	SRVAL	SRVAL	RAM(dir) =>datos	RAM(dir) =>pdatos
0	1	0	0	↑	dir	X	X	RAM(dir)	RAM(dir)	S/C	S/C
0	1	0	1	↑	dir	datos	Pdatos	S/C [a] RAM (dir)[b] datos[c]	S/C [a] RAM(dir)[b] pdatos[c]	RAM(addr) =>datos	RAM(dir) =>pdatos

GSR=Señal para Set y Reset globales

INIT=Valor especificado en el atributo INIT para los datos en memoria. Su valor por defecto es a ceros
SRVAL= Valor de salida tras activarse SSR, que coincide con el valor del atributo SRVAL, especificado por el usuario.

S/C= Sin cambios
dir=dirección de la RAM
RAM(dir)=Contenidos de la RAM en la dirección DIR
data=Entrada de datos en la RAM
pdata=Datos de paridad en la RAM

[a]WRITE_MODE=NO_CHANGE
[b]WRITE_MODE=READ_FIRST
[c]WRITE_MODE=WRITE_FIRST

Se pueden utilizar los atributos *INIT_XX* para especificar valores iniciales de los contenidos en memoria de una *RAMB16*. Esta inicialización se establece para cada *RAMB16_Sn* por medio de 64 valores hexadecimales (desde *INIT_00* hasta *INIT_3F*), cada uno de los cuales representa 16384 bits. Paralelamente, se pueden usar también los atributos *INITP_XX* para especificar valores iniciales en la memoria de paridad. Esta nueva inicialización se puede llevar a cabo mediante 8 atributos iniciales (desde *INITP_00* hasta *INITP_07*), compuestos por 64 valores hexadecimales, que conforman un total de 2048 bits; a través de los puertos configurados para 9, 18 o 36 bits. Si estos atributos (*INIT_XX* o *INITP_XX*) se quedan sin especificar, se les asigna como valor por defecto a ceros. A las cadenas de bits incompletas se les añaden ceros a la izquierda hasta completar el total del bus.

En las *placas Spartan-3, Virtex-II, Virtex-II Pro, y Virtex-II Pro X*, cada bit del registro de salida se puede inicializar a 0 o a 1. Esta inicialización se puede realizar por medio de dos atributos: *INIT* y *SRVAL*. El atributo *INIT* especifica la salida de este registro de salida cuando la FPGA se enciende (*power on*), mientras que el atributo *SRVAL* especifica la salida cuando se activa la señal de entrada SSR. Sus valores por defecto son a ceros.

En cuanto al modo de escritura, sólo comentar brevemente que el atributo *WRITE_MODE* controla los contenidos y la salida de la *RAMB16_SX*. Por defecto, su valor es *WRITE_FIRST*. Esto implica que si se realiza una escritura, en la salida se muestran los datos de la entrada. También se puede configurar el *WRITE_MODE* a *READ_FIRST* para que, cuando se realice una escritura, mostrar en la salida el contenido en memoria que había antes de que se

realizase dicha escritura. Asimismo, se puede configurar el *WRITE_MODE* a *NO_CHANGE* para que, cuando se realice una escritura, no se observen cambios en la salida de datos de la RAM.

Por último, comentar que se utilizará un tipo de *RAMB16_SX* u otro en función de las necesidades de la FIFO en cuestión, en términos de longitudes de palabra y/o número de entradas. Se ha observado que cuanto mayor tamaño de palabra tenga la memoria que se utilice, más recursos consume (en todos los casos se utiliza una BLOCK RAM, pero a mayor tamaño de palabra, se realiza un mayor número de interconexiones, con el consecuente consumo de recursos). Por ello, se han implementado un total de cuatro tipos de FIFOs diferentes, que utilizaremos en los siguientes contextos:

- ***FIFO3BITS*:** Utilizamos una *RAMB16_S4*, para la FIFO de reconfiguraciones.
- ***FIFO5BITS*:** Utilizamos una *RAMB16_S9,* para la FIFO de eventos.
- ***FIFO13BITS*:** Utilizamos una *RAMB16_S18*, en los módulos para las URs.
- ***FIFO30BITS*:** Utilizamos *RAMB16_S36*, en la tabla de tareas.

Así se consigue hacer un uso muy eficiente de los recursos disponibles. Por otro lado, la configuración de los parámetros de inicialización de las BLOCK RAMs es la siguiente:

- ***WRITE_MODE*** = Configurado a NO_CHANGE.
- ***Datos de paridad***: No utilizados.
- ***Contenido inicial de las FIFOs***: a ceros.
- ***GSR***: No utilizada
- ***SSR***: Reset activo a baja.

2) Implementación de las FIFOs

El manejo de una planificación de tareas dada exige por parte del sistema el uso de complejas estructuras de datos, como listas enlazadas o indexadas. En la figura 18 se puede observar un ejemplo de motivación.

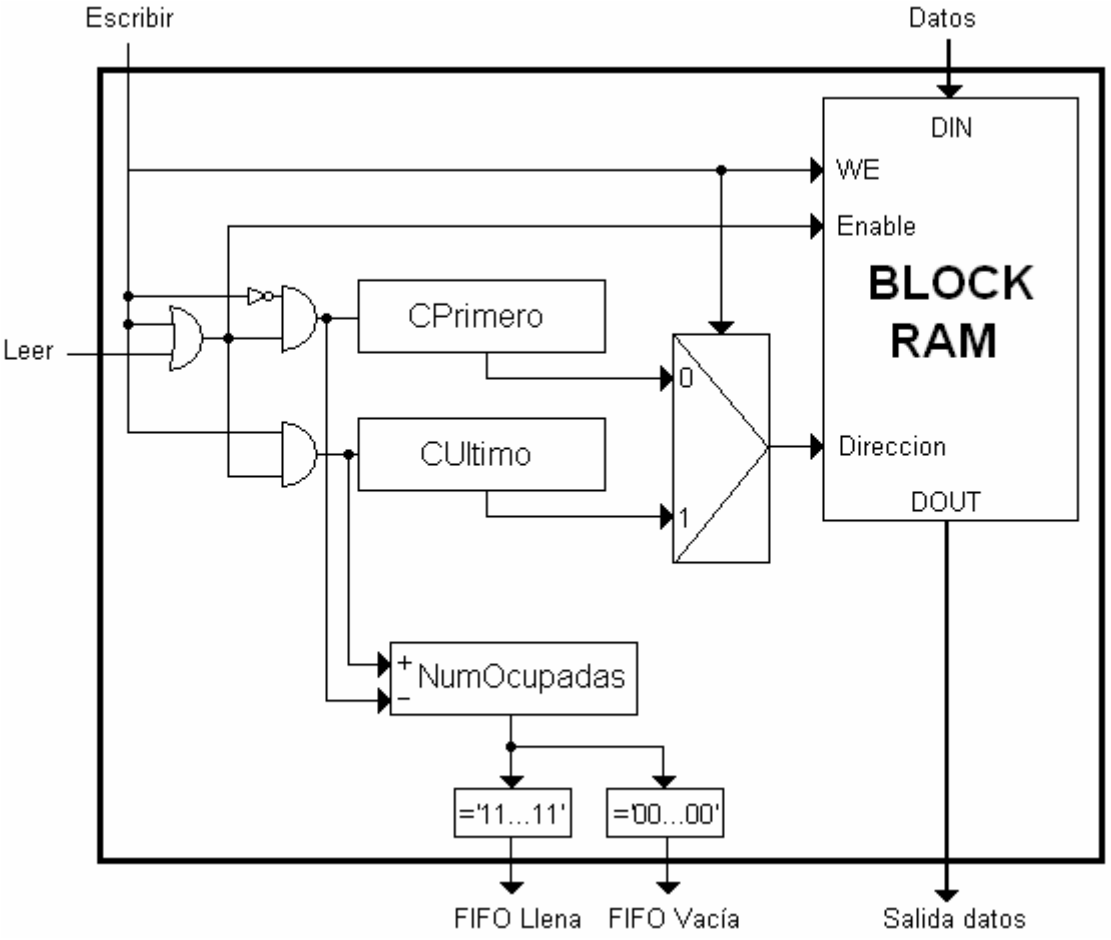
Esto ha obligado a implementar un HW que proporcione el soporte de almacenamiento necesario para llevar a cabo un manejo eficiente y transparente de operaciones de inserción, borrado y actualización de subtareas en listas enlazadas (figura 21). Este HW implementa la política de actualización FIFO (*First in, First out*), por lo que conceptualmente representa una cola cuyos elementos se insertan al final de la misma. Por otro lado, al realizar una operación de extracción de un elemento, se extrae el que esté en primer lugar en la cola. El número de entradas de las FIFOs es constante.

Las operaciones que soporta son las siguientes:

a) ***Inserción***: Se inserta un elemento al final de la cola. Si la FIFO estuviera llena, no se llevaría a cabo esta operación y se encontraría activa la señal "*FIFO llena*".
b) ***Extracción***: Se extrae el primer elemento de la lista. Si la FIFO se vaciara como consecuencia de una extracción de su único elemento en la misma, se activaría la señal "*FIFO vacía*". Si la FIFO ya estuviera vacía y se intentase realizar una extracción, ésta no se lleva a cabo con éxito y la señal "*FIFO vacía*" permanece activa.

Por otro lado, las FIFOs implementadas sólo tienen una entrada de datos, por lo que solamente se pueden realizar una lectura o escritura a la vez. No se permiten lecturas y escrituras simultáneas; y en el caso de que se intentase realizar dicha operación, se daría prioridad a las escrituras. (Se ha configurado el *WRITE_MODE* de la RAM a *NO_CHANGE*).

Figura 21. Esquema de las FIFOs desarrolladas



En cuanto a la implementación, su esquemático es el que aparece en la figura 21. Se utilizan dos contadores (*CPrimero* y *CUltimo*) para guardar la dirección de la primera y de la última posición de los datos en la tabla, respectivamente. Cuando se hace una operación de escritura, se escribe en la posición que indique el contador *CUltimo* (que apunta a la posición siguiente a la última ocupada) y éste se incrementa. Por otro lado, cuando se hace una operación de extracción, se realiza una lectura en la *BLOCK RAM* seleccionando como dirección la que indique *CPrimero* y se decrementa este contador. Así nos aseguramos que los datos están siempre en posiciones contiguas en la *BLOCK RAM*. También utilizamos un contador ascendente/descendente para saber cuántas posiciones hay ocupadas en la FIFO. (*NumOcupadas*). Si este contador está a cero, la FIFO estaría vacía, mientras que si este contador está a su máximo valor posible, la FIFO estaría llena.

La tabla asociativa

Es la tabla asociativa en donde se guarda la información de la(s) tarea(s) presentes en el sistema. Esta tabla se utiliza para supervisar las dependencias entre las subtareas de una tarea, conceptualmente representada mediante un grafo. Su tamaño es escalable en cuanto a número de entradas y número máximo de sucesores. Se proponen dos implementaciones distintas: como una sola tabla totalmente asociativa o como un conjunto de subtablas independientes totalmente asociativas.

Las operaciones que soporta son: *inserción, borrado y actualización,* y *comprobación*, que pasamos a describir a continuación:

a) ***Inserción***: Esta operación escribe la información de la subtarea en la tabla.

   Con la implementación como una única tabla totalmente asociativa, la ubicación donde se escriben los datos no es relevante. Para seleccionar dónde se guarda una subtarea se utiliza la red iterativa, que siempre indica la primera entrada libre de la tabla; y se que genera un error si la tabla está llena. Con este enfoque la operación puede realizarse en un ciclo de reloj, pero a medida que el tamaño de la tabla crece, el ciclo de reloj dismuye.

   Con la imlementación como un conjunto de subtablas asociativas, al igual que en la anterior, la ubicación donde se escriben los datos no es relevante. Para seleccionar dónde se guarda una subtarea se pregunta secuencialmente a cada subtabla si puede ubicarse en ella hasta que se encuentre una ubicación disponible. Para ello nos ayudaremos de la red iterativa descrita anteriormente, que nos dirá que la subtabla se encuentra llena o en caso contrario, indica la primera entrada libre de la subtabla. Con este enfoque la operación puede llegar a tardar N ciclos de reloj, donde N es número de subtablas existentes, pero sin embargo, se consigue que al aumentar el número de subtablas disponibles, el ciclo de reloj permanezca constante.

b) ***Borrado y actualización***: Esta operación se realiza cuando una subtarea termina su ejecución. La operación elimina esta subtarea de la tabla, con lo que ésta puede ser usada en el futuro para ubicar otras subtareas. A continuación, se realiza una actualización de todas sus dependencias, consistente en disminuir en uno el número de sucesores de cada uno de sus predecesores. Esto es posible gracias a que en la entrada de la tabla guardamos la información acerca de todos los identificadores de todos predecesores; con lo que la actualización consistirá en buscar cada uno de estos predecesores en la tabla y disminuir en uno su número de sucesores. Queda claro, por tanto, que el objetivo de la actualización es garantizar en todo momento que la información que contiene la tabla sea correcta. Esta operación tiene una complejidad de Θ(N), donde N es el número de sucesores a actualizar.

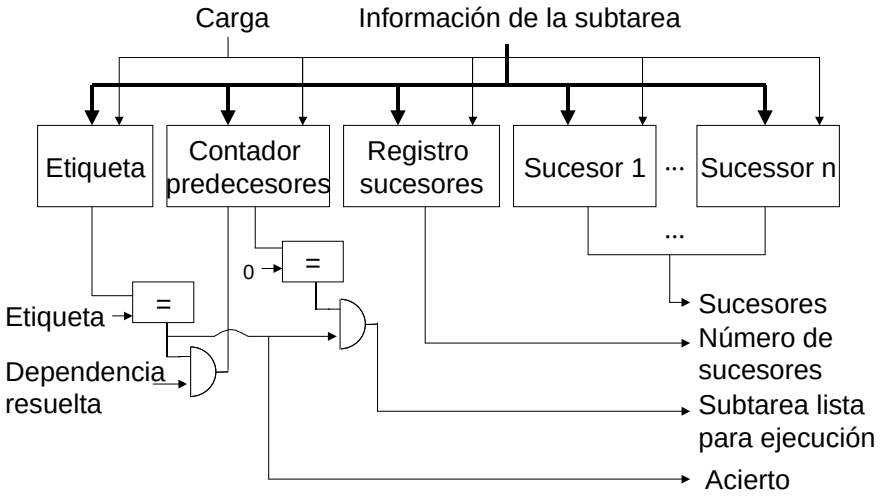
c) ***Comprobación***: Esta operación consulta si una subtarea dada está lista para comenzar su ejecución, es decir, si están resueltas todas sus dependencias. Esto se realiza en un sólo ciclo de reloj, introduciendo la etiqueta de la subtarea como entrada de la tabla y leyendo la salida *subtarea preparada* en el siguiente ciclo. Una subtarea está preparada si su contador de predecesores es cero

Estructuralmente, la tabla está compuesta por una serie de entradas unidas entre sí mediante una red iterativa (que sirve para decidir en qué entrada se guardará una nueva subtarea que entre en la tabla). Alternativamente, existe la posibilidad de implementarlo como un conjunto de subtablas más pequeñas, de forma que las inserciones en cada subtabla se gestionen por separado. Esta modificación reduce el retardo del sistema aunque aumenta el número de ciclos necesarios para cada inserción. Asimismo, existe también un controlador que gestiona las distintas operaciones que la tabla permite realizar.

1) Entrada de la tabla

Consiste en un módulo básico en el que se guarda toda la información relacionada con una misma subtarea: su identificador, el número de predecesores, el número de sucesores, así como los identificadores de estos sucesores. En la figura 22 se puede observar su esquemático.

Figura 22. Entrada de la tabla de dependencias de subtareas



El registro "*etiqueta*" guarda el identificador de la subtarea que se encuentra en la entrada. El contador de predecesores y el registro de sucesores guardan la información acerca del número de predecesores y sucesores, respectivamente; y que será muy útil para gestionar las dependencias correctamente. Por otro lado, los registros "*Sucesor 1*", …, "*Sucesor n*" guardan los identificadores de los sucesores de la tarea que esté en esa entrada. El resto del HW sirve para generar señales de control útiles para la correcta gestión de las dependencias: "*Subtarea lista para ejecución*" indica si la subtarea presente en la entrada está lista para su ejecución (equivalente a que su número de predecesores sea 0), y la señal "*acierto*" indica si la subtarea de la entrada es la que se pide desde la entrada "*Etiqueta*".

2) Red iterativa

Consiste en una red formada por una serie de celdas individuales conectadas entre sí y cada una de las cuales está a su vez conectada con una entrada de la tabla asociativa. En la figura 23 se puede observar su esquemático.

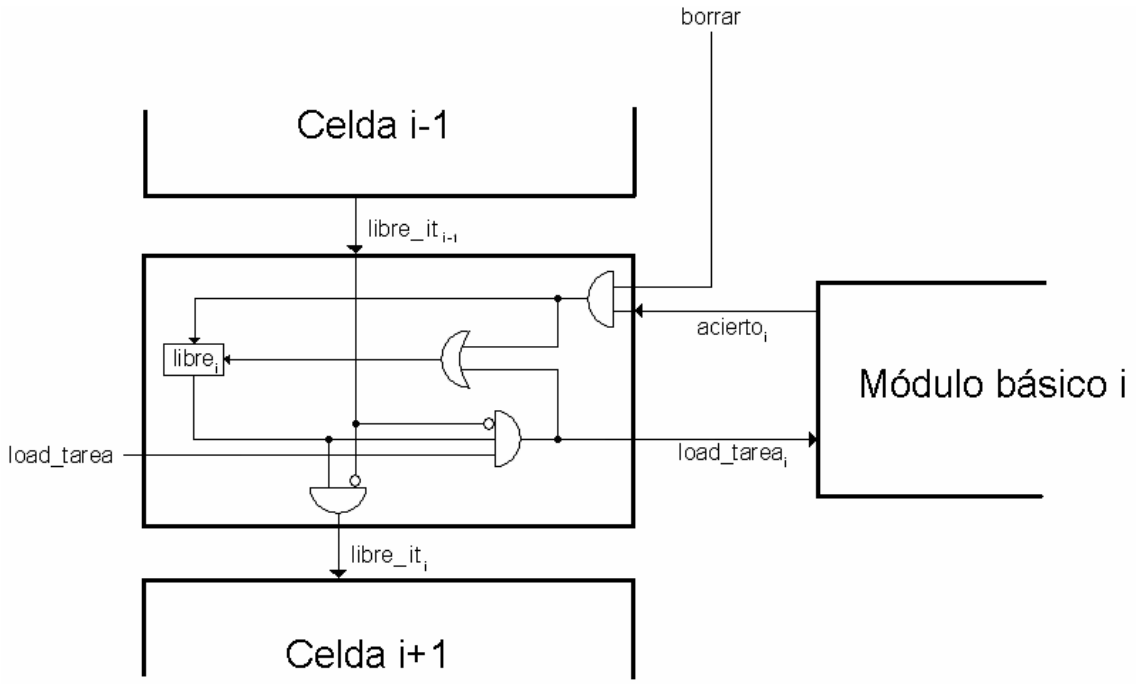
Esta red se utiliza para saber cuál es la primera entrada de la tabla que se encuentra libre ("$load\_tarea_i$"); y esta información es necesaria cuando se hace una inserción de una subtarea en la tabla.

Se trata de una red iterativa porque existe una señal (*libre_it*) que se van pasando unas celdas a otras. Su significado es el siguiente:

$$libre\_it_{i+1} = \begin{cases} 1 & si \;\; libre\_it_i = 0 \;\; AND \;\; \forall x \in [0..i-1] \; libre_x = 0 \\ 0 & en \;\; caso \;\; contrario \end{cases}$$

De esta forma, si hay varias posiciones libres en la tabla, sólo en la primera de estas celdas (empezando desde el subíndice 0) se activará $load\_tarea_i$, por lo que nos aseguramos que una subtarea se cargará en la primera posición libre de la tabla.

Figura 23. Red iterativa



Cada celda de esta red contiene un registro de un bit que indica si la entrada asociada a dicha celda está ocupada por una subtarea (1) o no (0). Por tanto, la actualización de su contenido sólo tendrá lugar en dos situaciones:

1) Cuando se cargue una nueva subtarea en el módulo básico asociado. Esto ocurre cuando la señal *load_tarea*$_i$ está activa, la celda está libre y la celda anterior no lo está. Si *load_tarea*$_i$ se activa, se desencadenaría la carga de la subtarea en el módulo básico correspondiente y se activaría la señal de carga del registro. Lo que se carga en este caso en el registro es un 0, ya que la señal *borrar* no está activa.
2) Cuando se pretenda eliminar esa subtarea. Esto ocurre cuando la señal *borrar* está activa y cuando la subtarea que se quiere borrar está en el módulo básico asociado (*acierto*$_i$ es 1). Se pondrían a 1 simultáneamente la señal de carga del registro y su entrada de datos, y se cargaría un 1.

Las operaciones que hacen uso del HW de la red iterativa tienen un tiempo de ejecución de un solo ciclo de reloj. Recordar que en el caso de la implementación con un conjunto de subtablas, es un ciclo por cada consulta (cada uso de la red iterativa de cada subtabla).

3) HW de control

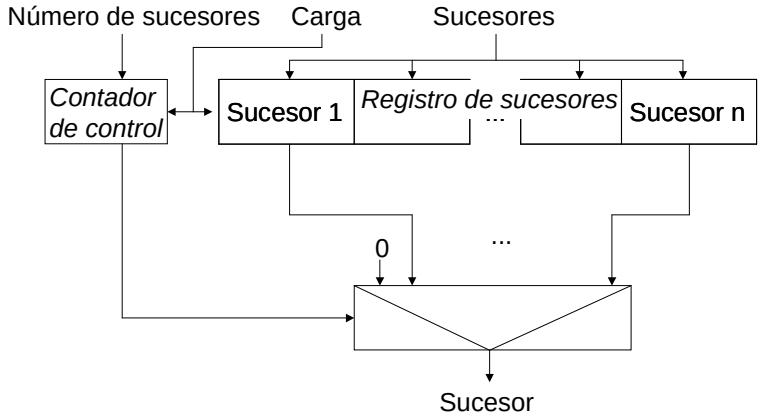
Para que las operaciones que soporta la tabla se puedan llevar a cabo correctamente es necesario cierto HW de control. Este HW consta de:

a) Por un lado, una circuitería combinacional que garantiza que la operación de *borrado y actualización* se lleve a cabo correctamente. En la figura 24 se puede observar su estructura.

   Disponemos de un contador, un registro de sucesores y un multiplexor. En el contador y en el registro de sucesores se guardan el número de sucesores y los

identificadores de los sucesores que tiene la tarea cuyas dependencias queremos actualizar, respectivamente.

Figura 24. Soporte HW para la operación borrado y actualización



Su funcionamiento es el siguiente: se empieza cargando la información correspondiente en el contador y en el registro de sucesores. Después, se actualiza secuencialmente cada sucesor, operación que se realiza en un ciclo de reloj para cada uno. En cada ciclo se realizan tres operaciones: uno de los sucesores se selecciona en el registro de sucesores usando el *contador de control* y el multiplexor, se activa la entrada de la tabla asociativa *dependencia resuelta* y el *contador de control* se decrementa. Cuando el contador de control toma el valor cero la operación finaliza. Por su parte, cuando la tabla detecta que la señal *dependencia resuelta* está activa, decrementa el contador de predecesores de la subtarea correspondiente.

b) Por otro lado, tenemos un controlador que garantiza que las operaciones anteriormente mencionadas se lleven a cabo correctamente, gobernando las señales de control del HW combinacional de este módulo de manera apropiada.

4) Esquemático a alto nivel

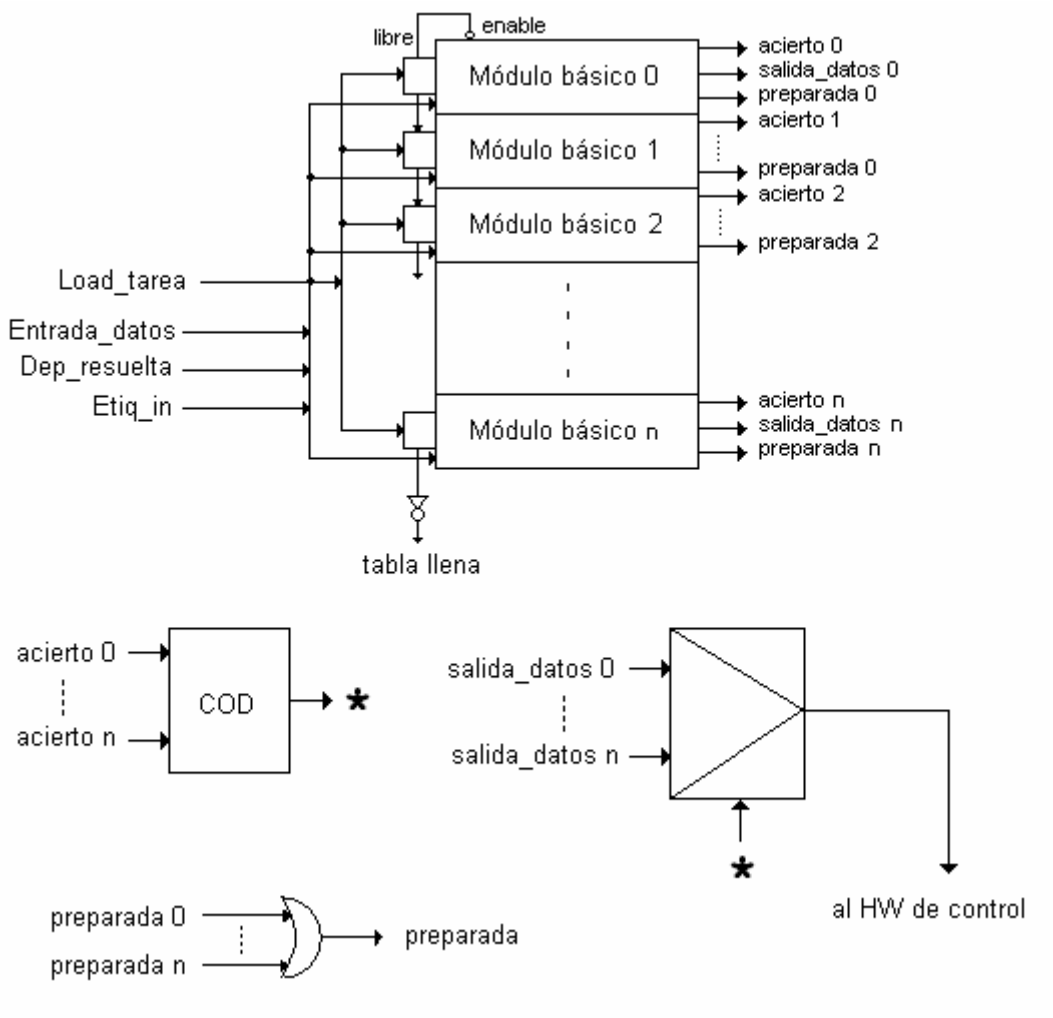
1) Primera implementación – Una única tabla totalmente asocitativa

Todos los módulos anteriormente explicados y debidamente interconectados componen la tabla de tareas, cuyo esquemático a alto nivel es el que indica la figura 25.

Vemos que la red iterativa está estrechamente relacionada con las celdas básicas de la tabla asociativa, pues cada celda de la red está conectada con una entrada de la tabla. Las entradas "*Entrada_datos*", "*Dep_resuelta*" y "*Etiq_in*" entran directamente en las celdas básicas de la tabla asociativa, mientras que "*Load_tarea*" entra *también* en la red iterativa.

Por otro lado, el codificador sirve para determinar la posición en que se encuentra la celda cuyo contenido coincide con la entrada "*Etiq_in*". Posteriormente, mediante un multiplexor se selecciona la salida de datos correspondiente a la tarea seleccionada y estos datos se utilizan en el siguiente HW de control, cuya función es garantizar que se realice la actualización de las dependencias en cada uno de sus sucesores. Esta actualización se realizará en la operación "*borrado y actualización*", explicada con detalle en el apartado "*Operaciones*".

Figura 25. Tabla de tareas

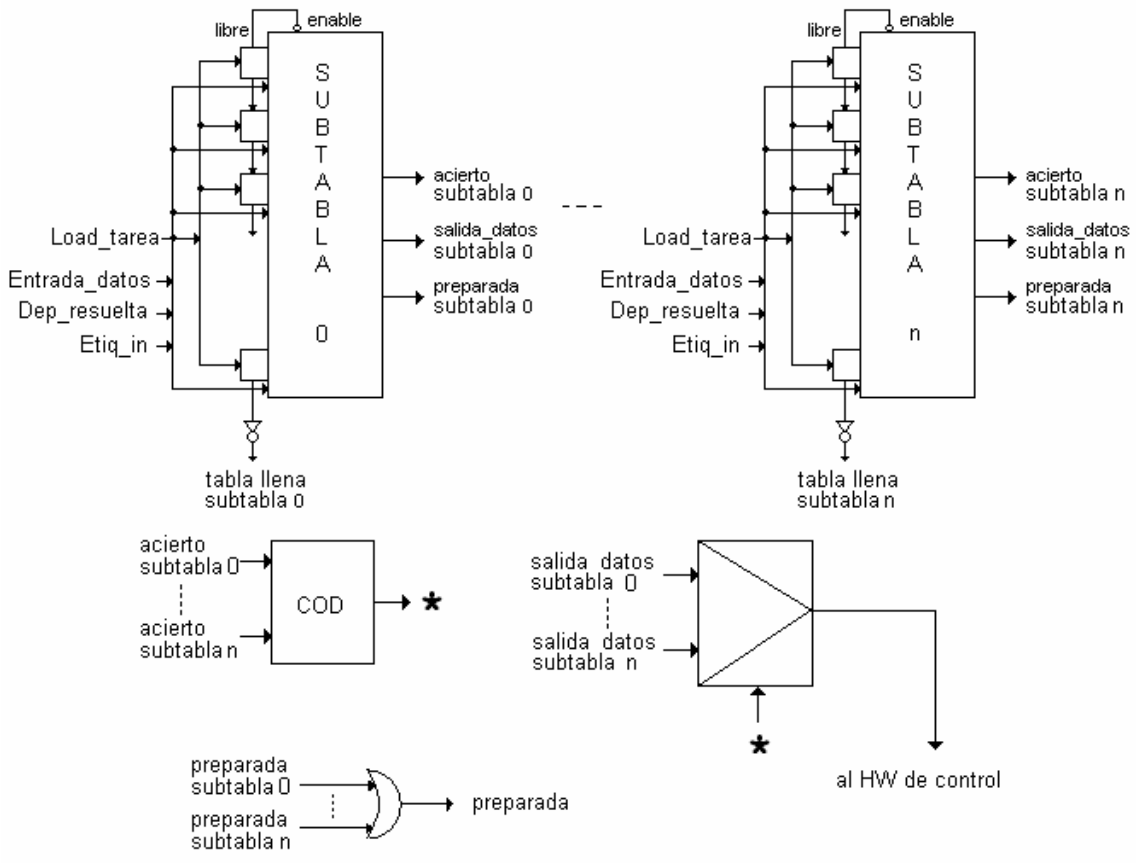


2) Segunda implementación – Conjunto de subtablas asociativas

Como puede verse en la figura 26, se trata de un conjunto de subtablas asociativas como las explicadas anteriormente y una lógica combinacional para seleccionar las señales de salida correspondientes a la subtabla donde está ubicada la subtarea que se esté tratando.

Esta implementación resulta ventajosa, ya que el módulo crítico del sistema es la red iterativa que nos dice cuál es la primera entrada libre de la tabla o si ésta se encuentra llena. De esta forma, al quedar particionada la red iterativa, se consigue que el ciclo de reloj permanezca constante a media que aumenta el número de subtablas asociativas del gestor. Sin embargo, mientras las operaciones de *comprobación* y *borrado y actualización* tienen el mismo comportamiento que anteriormente, se van a sufrir penalizaciones en la operación de *inserción*. De forma consecutiva, comprobamos si la subtarea cabe en la subtabla o si ésta se encuentra llena, hasta que finalmente se encuentre una subtabla con una entrada libre o si todas se encuentran llenas, en este caso, generamos una señal de error. Con este enfoque la operación puede llegar a tadar N ciclos de reloj, donde N es número de subtablas existentes, pero sin embargo, conseguimos que al aumentar el número de subtablas disponibles, el ciclo de reloj permanezca constante.

Figura 26. Tabla de subtablas de tareas

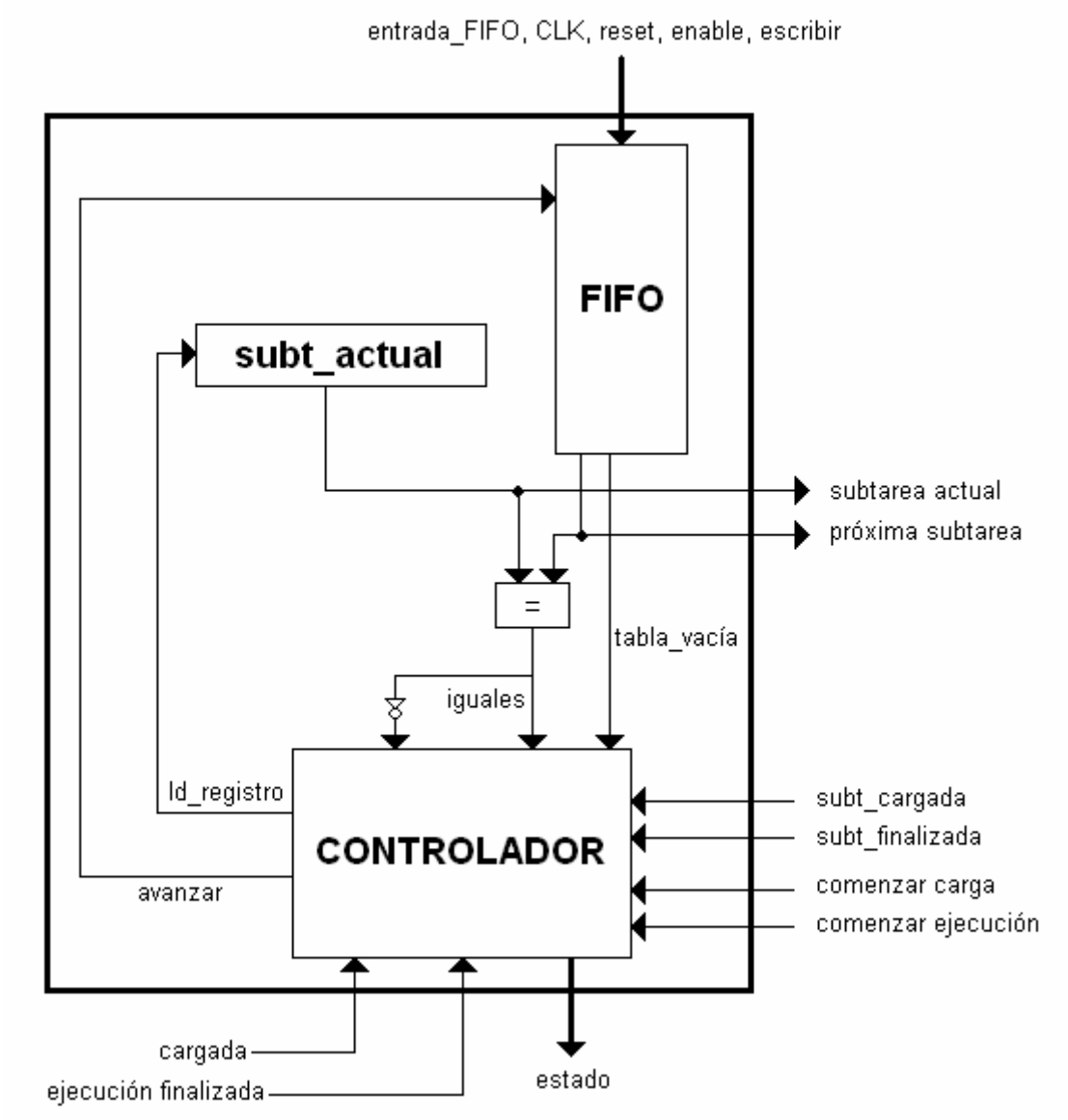


El módulo para las URs

Este módulo se encarga de caracterizar cada una de las unidades reconfigurables del planificador. Su misión es proporcionar información a la unidad de control sobre el estado en el que se encuentra cada una de las unidades reconfigurables. (Para más información acerca de este módulo consultar el apartado dedicado a la *unidad de control*.)

En el marco global del sistema, este módulo está continuamente funcionando y sincronizándose únicamente con el árbitro para gestionar las escrituras a la FIFO de eventos. A grandes rasgos, lo que hace es extraer subtareas de su FIFO local (no necesita ocuparse de cómo llegan ahí esas subtareas) y tratarlas secuencialmente. Para cada una de ellas, debe ocuparse de simular su reconfiguración y/o ejecución. Si no hay subtareas en su FIFO local, no hace nada. Su esquemático es el que se puede ver en la figura 27.

Figura 27. Módulo para las unidades reconfigurables



Las tareas que realiza son:

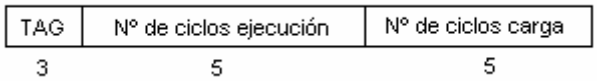
1- Responder a las órdenes de comienzo de carga y comienzo de ejecución. En esos casos, este módulo simula la carga y/o la ejecución de la subtarea que se encuentre actualmente en la unidad.
2- Generar los eventos apropiados:
   a. Cuando se termina de ejecutar una subtarea se genera el evento “subtarea ejecutada”. El código de evento es: “10”
   b. Cuando se termina de reconfigurar una subtarea o cuando una subtarea es reutilizada, el evento que se genera es “subtarea reconfigurada”. El código de evento es: “01”.

Estos eventos se escriben en la FIFO de eventos. Una situación problemática que podría suceder es que varias unidades reconfigurables intentasen escribir simultáneamente en dicha FIFO. Para que esa situación no represente problema alguno, existe un árbitro externo que garantiza el acceso exclusivo a las operaciones de escritura en la FIFO, a la vez que organiza las peticiones para que ninguna de ellas se quede sin atender. Para más información acerca del árbitro consultar el apartado dedicado al *árbitro*.

Vemos que este módulo consta de los siguientes componentes HW:

1- Un registro en el que se guarda la subtarea actual de la unidad.
2- Una FIFO en la que se guardan las subtareas que están planificadas para ser ejecutadas en esa unidad. En concreto, usamos una *FIFO13BITS*, ya que las palabras que guarde tendrán el formato que se indica en la figura 28. Necesitamos saber cuántos ciclos tarda la ejecución y la reconfiguración de la tarea para poder así generar correctamente los eventos correspondientes.

Figura 28. Formato de las palabras en la FIFO de las URs



Para su implementación, se ofrece la posibilidad de utilizar una Block RAM común a todas o a varias URs, con el espacio de memoria repartida de forma equitativa entre ellas. Dado que los tiempos de ejecución y de reconfiguración de las distintas subtareas serán diferentes entre ellas, la probabilidad de que se produzcan dos o más escrituras o lecturas al mismo tiempo es baja. Pero dado que puede ocurrir, se ha añadido un árbitro con prioridad fija para resolver el problema. De esta forma se ahorran recursos a costa de sufrir penalizaciones con una probabilidad de aparición baja.

3- Un HW de control que gestiona los cambios de estado activando las señales correspondientes, así como las señales de salida del módulo. Este HW de control está compuesto por el comparador y el controlador, cuyo diagrama de estados se puede observar en la figura 29.

Este diagrama de estados es una máquina de Moore. Sus estados son los siguientes:

"***lee_fifo***": La unidad reconfigurable está en este estado cuando está intentando leer de su FIFO de reconfiguraciones. Permanece en este estado mientras la tabla esté vacía. En el momento en que se detecte que hay al menos una subtarea en dicha FIFO, pasa al estado "*finalizada*".

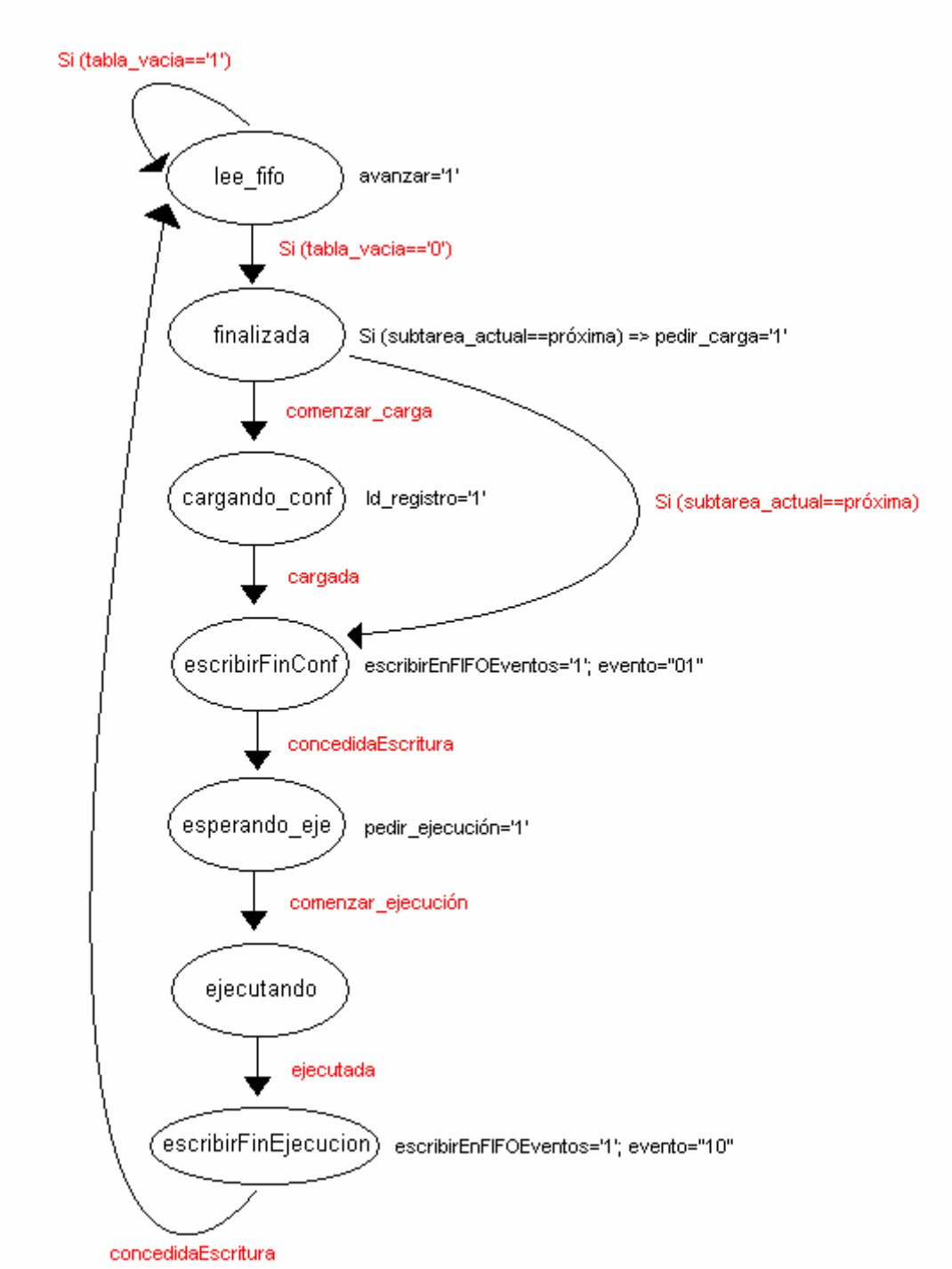
"***finalizada***": La unidad reconfigurable está en este estado cuando se ha terminado de extraer una subtarea de la FIFO de reconfiguraciones. En este estado se decide si se debe o no pedir la carga de esta subtarea recién extraída (se debe reutilizar cuando es igual a la que había inmediatamente antes en la unidad). En caso de que haga falta reconfigurar (porque no coincida con la inmediatamente anterior) se pide la carga (*pedir_carga*=='1') y, cuando se le ordene la carga (*comenzar_carga*), comience la reconfiguración. En caso contrario, pasa directamente al estado "*escribirFinConf*".

"***cargando_conf***": La unidad reconfigurable está en este estado mientras se esté llevando a cabo la reconfiguración de la subtarea. Es en este estado cuando se carga la subtarea en el registro de la unidad reconfigurable. El número de ciclos durante los que la unidad se encuentre en este estado dependerá del tiempo de reconfiguración, tiempo que viene incluido entre la información que se proporciona de la subtarea. Tan pronto como se recibe la confirmación de que la subtarea ha sido cargada (*cargada*=='1') pasa al estado "*escribirFinConf*".

"***escribirFinConf***": La unidad reconfigurable está en este estado cuando se ha finalizado la reconfiguración de una subtarea o cuando se acaba de decidir que una subtarea debe ser reutilizada. En este estado se genera el evento "*subtarea reconfigurada*". Este evento se genera tanto si una subtarea es reconfigurada como si es simplemente reutilizada. El motivo por el que se genere el mismo evento en dos situaciones diferentes es que ambas situaciones son equivalentes desde el punto de vista del gestor de los eventos. La unidad permanece en este estado el tiempo que haga falta hasta que se le concede la escritura en la FIFO de eventos; porque, como ya se dijo anteriormente, puede ocurrir que varias unidades estén funcionando en paralelo y quieran escribir simultáneamente en dicha FIFO, con el consecuente

problema de una escritura concurrente que la FIFO no soporta. Para garantizar el acceso exclusivo a estas operaciones de escritura debe existir un árbitro y es precisamente ese árbitro el que informa a los módulos UR acerca de las concesiones.

Figura 29. Diagrama de estados para el control de los módulos de las URs



"***esperando_eje***": La unidad reconfigurable está en este estado mientras se encuentre a la espera de comenzar la ejecución de su subtarea. Cuando se reciba desde el exterior la señal de que dicha ejecución debe comenzar, pasa al estado "*ejecutando*".

"***ejecutando***": La unidad reconfigurable está en este estado mientras la subtarea que contiene se esté ejecutando. Permanece en este estado tantos ciclos como tarde en llevarse a cabo dicha ejecución; y, cuando finaliza, activa la señal "*ejecutada*" y pasa al estado "*escribirFinEjecucion*".

"escribirFinEjecucion": La unidad reconfigurable está en este estado cuando se ha finalizado la ejecución de una subtarea. En este estado se genera el evento "*subtarea ejecutada*". Al igual que ocurría con el estado "*EscribirFinConf*", la unidad permanece en este estado el tiempo que haga falta hasta que se le concede la escritura del evento ahora generado en la FIFO de eventos. Cuando se recibe una confirmación de que dicho evento ha sido escrito con éxito (*concedidaEscritura*= ='1'), pasa al estado "*lee_fifo*".

La FIFO de reconfiguraciones

Disponemos de una FIFO en la que se guardan qué subtareas deben ser reconfiguradas en el sistema y en qué orden. Es, por tanto, en esta FIFO donde que se guarda la información acerca de la planificación que debe seguir el sistema. En concreto, utilizamos una *FIFO3bits*, ya que guardamos una lista de identificadores de subtareas; y dichos identificadores tienen 3 bits.

Esta FIFO se actualiza únicamente cuando entra una nueva tarea, momento en el cual se escriben en ella (en un determinado orden) todas las reconfiguraciones que han de producirse. Posteriormente, se irán extrayendo subtareas de esta FIFO y se les dará un tratamiento adecuado: buscar en qué unidad está; y cuando esta unidad esté en estado "finalizada", ordenar una ejecución (si tiene lugar una reutilización de esta subtarea en la UR) o una reconfiguración (en caso contrario). Este tratamiento tiene lugar en la unidad de control.

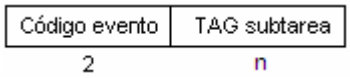
Sus características tecnológicas son las de una FIFO ya explicada anteriormente.

La FIFO de eventos

Disponemos de una FIFO de eventos para guardar y así poder gestionar los eventos que se producen en el sistema en tiempo de ejecución. En concreto, utilizamos una *FIFO5bits*, ya que el formato de palabras que se usarán en este caso es el que se indica en la figura 30.

Los eventos que se pueden producir en el sistema y, que, por tanto, son susceptibles de ser guardados aquí, se pueden generar desde los módulos que caracterizan a las URs o desde el cargador de tareas; este último caso sólo ocurre cuando entre una nueva tarea en el sistema.

Figura 30. Formato de las palabras en la FIFO de eventos



"nueva tarea": Este evento se genera cuando entra una nueva tarea en el sistema. Es el evento que menos veces se genera en el sistema; sólo una vez por tarea recibida. Su código binario es "11".

"fin de ejecución de la subtarea i": Este evento se genera cuando se termina de ejecutar una subtarea. Como una subtarea se ejecuta dentro de una UR, será el módulo que caracteriza a su UR el que genere este evento. Su código binario es "10".

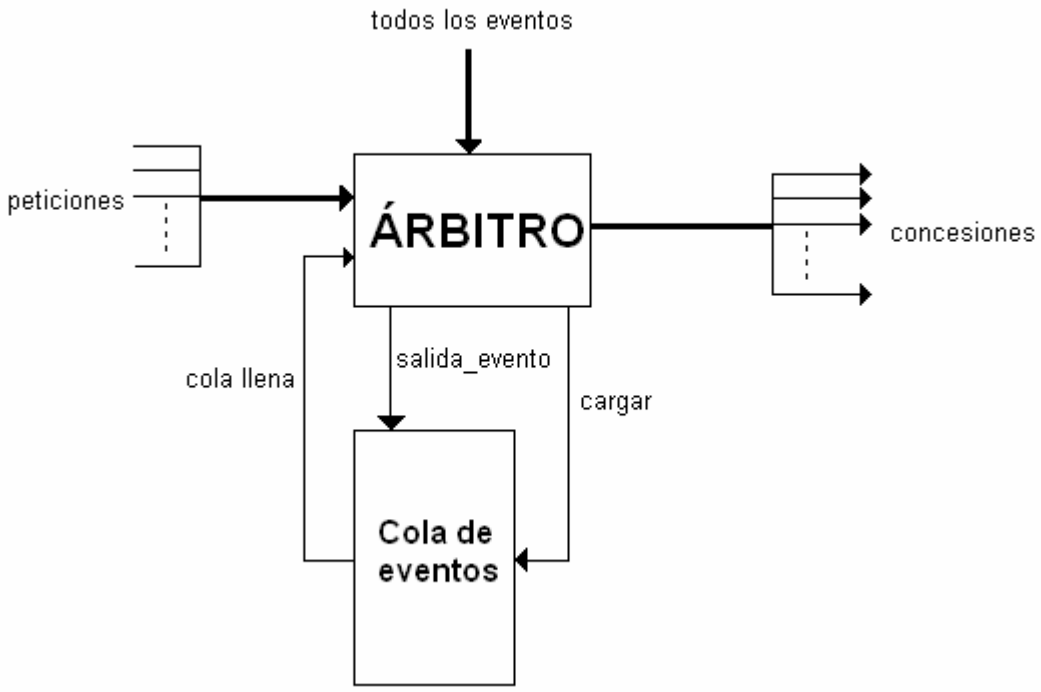
"fin de reconfiguración de la subtarea i": Este evento se genera cuando se termina de reconfigurar una subtarea. Se genera desde los módulos que caracterizan a las URs. Su código binario es "01".

"subtarea i reutilizada": Este evento se genera cuando se termina de ejecutar una subtarea. Se genera desde los módulos que caracterizan a las URs. A efectos prácticos, este evento y "*fin de ejecución de la subtarea i*" son equivalentes debido a que a ambos se les da el mismo tratamiento en la unidad de control. Por tanto, en las situaciones en las que se deba generar este evento, se generaría "*fin de reconfiguración de la subtarea i*", con la consecuente simplificación de la gestión de los eventos.

El árbitro

Se acaba de mencionar que en el sistema propuesto se lleva a cabo una gestión de eventos, consistente en su generación, almacenamiento en una FIFO y posterior tratamiento. Debido a la naturaleza imprevisible (al menos desde el punto de vista del sistema) de estos eventos, es posible que se generen simultáneamente dos o más eventos y que, por tanto, se intenten escribir simultáneamente en la FIFO de eventos (entendemos "simultáneamente" por "en el mismo ciclo de reloj"). Esta es una situación problemática porque las FIFOs utilizadas no soportan dos o más escrituras simultáneas, al tener un solo puerto de escritura. Por tanto, es necesario garantizar la exclusión mutua en los accesos este puerto de escritura para que no quede ningún evento sin capturar. Para ello, se ha diseñado un árbitro (figura 31) que recibe todos los eventos que se generan en el sistema y establece una política de prioridades fijas de forma que se garantiza que sólo se intenta escribir en esta FIFO en un instante de tiempo dado.

Figura 31. Árbitro de interconexión con la FIFO de eventos



Este árbitro recibe todos los eventos en *eventos[1..n]*, (en el que *eventos[i]* es un código de dos bits) que se pueden producir en el sistema y *n* señales de peticiones (*peticiones[1..n]*) procedentes de cada uno de los módulos que los generan. Si *peticiones[i]* está activa, se hace una petición para que *eventos[i]* se escriba en la FIFO. Por otro lado, la señal de salida *concesiones[1..n]* indica las concesiones que se realizan: Si *concesiones[i]* está activa, se permite la escritura de *eventos[i]* en la FIFO; en caso contrario, se prohíbe dicha escritura.

Obviamente, sólo una de las *concesiones[i]* posibles estará activa en el mismo instante de tiempo.

El árbitro da la mayor prioridad a la escritura del evento "*nueva_tarea*"; seguido de los generados por las URs, de más significativa a menos significativa. (Se supone que las URs están identificadas por el árbitro de la forma *UR[1..n]*). Por tanto, su comportamiento es similar a un codificador de prioridad.

Además, el árbitro también controla las lecturas de la FIFO de eventos, dándoles mayor prioridad que a las escrituras.

La unidad de control

La unidad de control extrae los eventos de la cola y lleva a cabo las acciones apropiadas. Su pseudocódigo se describe en la figura 32.

Cuando un evento "*fin de ejecución*" se procesa, este módulo actualiza las dependencias guardadas en la tabla asociativa. Entonces, si el circuito de reconfiguración está libre, trata de empezar una reconfiguración. Si el sistema usa la opción basada en una planificación, leerá la FIFO de reconfiguración, y comprobará si es posible comenzar la reconfiguración correspondiente. Finalmente, la unidad de control comprobará si cualquiera de las subtareas que actualmente están cargadas puede comenzar su ejecución.

Para los eventos "*fin de reconfiguración*" y "*subtarea reutilizada*" la unidad de control comprobará si la subtarea que ha sido cargada puede comenzar su ejecución.

Finalmente, para el evento "*nuevo grafo*", la unidad de control leerá los datos desde el buffer de entrada y los almacenará en la tabla asociativa y en las respectivas FIFOs. Tras esto, si el circuito de reconfiguración está libre, comprobará si es posible comenzar a cargar una de las nuevas subtareas.

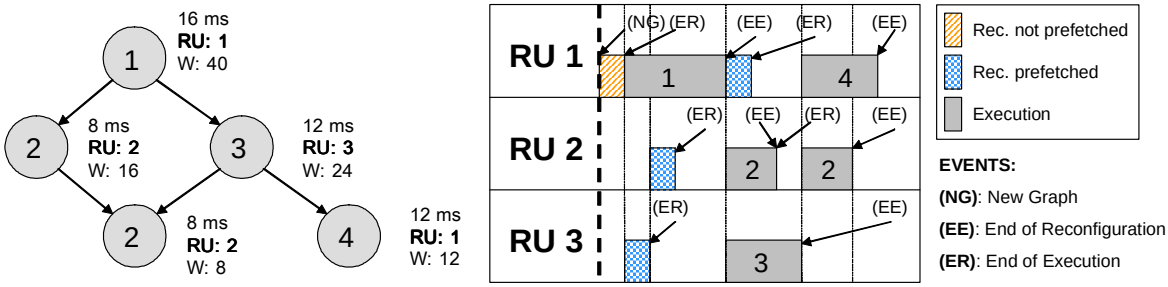
Figura 32. Pseudo-código de la unidad de control

```
Gestión de eventos:
 CASE evento IS:
  fin_de_ejecucion:
    actualizar_tabla
    buscar_una_reconfiguración()
    FOR i=0 hasta número de URs
       IF subtarea_preparada
            comenzar_ejecución()
  fin_de_reconfiguración;
    IF subtarea_preparada
        comenzar_ejecución()
        buscar_una_reconfiguración()
  nuevo_grafo:
    actualizar_tabla
    actualizar_FIFOs_UR
    IF politica_precarga=basado_planifica
           actualizar_FIFO_reconfig
           buscar_una_reconfiguración()
```

EJEMPLO DE EJECUCIÓN DE UNA TAREA

En esta sección se describe paso a paso la ejecución del grafo de subtareas mostrado en la figura 33. Esta figura representa a su vez la planificación seleccionada.

Figura 33. Ejemplo de ejecución de un grafo de subtareas



Inicialmente, el procesador enviará la información referente al grafo y el método de planificación elegido a nuestro gestor. Cuando la información esté correctamente almacenada en un buffer de entrada, el procesador generará el evento *nuevo grafo*. Tan pronto como la unidad de control procese este evento, toda la información será almacenada en la tabla asociativa y en las FIFOs. Además, la unidad de control intentará empezar una reconfiguración. Si el sistema está usando una planificación *basada en prebúsqueda* y la secuencia de reconfiguración elegida por el planificador es: 1-3-2-4, la unidad de control comenzará la reconfiguración de la subtarea 1.

Cuando la subtarea 1 termine su reconfiguración, la UR1 generará un evento fin de reconfiguración.

Ahora, la unidad de control comprobará si la subtarea 1 puede empezar su ejecución. En este caso, no hay dependencias no resueltas, por lo tanto comenzará justo después de que la reconfiguración termine.

Además, la unidad de control comenzará la siguiente reconfiguración ya que el circuito de reconfiguración está libre y en la UR3 no hay ninguna tarea en ejecución.

Cuando esa reconfiguración termine, generará un nuevo evento fin de reconfiguración, y la unidad de control comenzará la reconfiguración de la subtarea 2. Además, se comprobará a su vez si la subtarea 3 puede comenzar su ejecución, pero en este caso la tabla asociativa indicará que hay una dependencia aún no resuelta.

Cuando esta reconfiguración finalice, un nuevo evento será procesado, pero esta vez, no dará comienzo una nueva reconfiguración, ya que las subtareas nunca serán reemplazadas hasta que hayan sido ejecutadas. Tampoco se comenzará la ejecución de ninguna subtarea, ya que las subtareas 2 y 3 tienen dependencias no resueltas.

Cuando la subtarea 1 finalice su ejecución se generará un nuevo evento, que actualizará las dependencias e intentará comenzar una reconfiguración, en este caso de la subtarea 4. Además comenzará la ejecución de las subtareas 2 y 3.

Cuando la reconfiguración de la subtarea 4 finalice, se comprobará si puede comenzar su ejecución, pero en este caso no es posible ya que existen dos dependencias no resueltas.

El siguiente evento es el final de la ejecución de la subtarea 2. De nuevo, la tabla asociativa será actualizada y el sistema intentará comenzar una nueva reconfiguración, en este la reconfiguración de la subtarea 2. Pero como esta tarea ya está cargada, el sistema la reutilizará, generando solamente un evento tarea reutilizada. Este evento se procesará a

continuación, tratándose de comenzar la ejecución de la subtarea 2, pero dado que todavía existe una dependencia por resolver, la ejecución no podrá comenzar.

Cuando la subtarea 3 finalice, el sistema actualizará las dependencias e identificará que las subtareas 2 y 4 pueden empezar sus ejecuciones. Finalmente, cuando estas subtareas terminen, el gestor generará una interrupción para informar al procesador que la ejecución del grafo ha sido completada.

RESULTADOS EXPERMIENTALES

En esta sección, se evaluará el rendimiento de nuestro gestor y el área necesaria para implementarlo. Para este fin se ha implementado el circuito en una FPGA *VIRTEX-II PRO xc2rp30* utilizando *ISE 9.1i* y *EDK 9* como plataforma de desarrollo.

En los diseños realizados se ha empleado en todos ellos genéricos para que sean ajustables, como por ejemplo, el tamaño de la tabla asociativa, el máximo número de sucesores por tarea y el número de URs en el sistema. Puesto que estos parámetros influirán en el rendimiento y el coste del sistema, se evaluará un pequeño gestor con una tabla asociativa de tan sólo ocho entradas y cuatro URs, y más adelante se estudiará cómo evolucionan el rendimiento y el coste cuando el sistema aumenta de tamaño.

Como primer experimento se ha ejecutado el gestor propuesto para planificar el grafo de subtareas de la figura 33, y algunos otros grafos de tareas correspondientes a tres aplicaciones multimedia: un decodificador JPEG, una aplicación de reconocimiento de patrones y un codificador MPEG. La tabla 4 muestra los resultados de esas tres ejecuciones. La columna de retardo añadido incluye todos los retrasos generados en la ejecución del grafo de subtareas debido a todos los cálculos realizados por nuestro gestor de ejecución.

Tabla 4. Evaluación del rendimiento

Tarea	Número de subtareas	Retardo añadido	Penalización
Figura 33	5	400 ns	-66%
JPEG	4	450 ns	-75%
Patrón rec.	7	420 ns	-80%
MPEG	5	580 ns	-70%

La columna penalización contiene el porcentaje de disminución de las penalizaciones generadas por las reconfiguraciones gracias a la técnica de prebúsqueda. Para explicar los resultados utilizaré el grafo de la figura 33. En este caso, sólo cuatro de los eventos serán manejados por el gestor han introducido retardos. Estos eventos y los retardos pueden ser visualizados en la tabla 5. Estos eventos han generado retardos porque todos están trabajando con subtareas que pertenecen al camino crítico del sistema. El retardo total debido a la gestión del grafo de subtareas y la aplicación de las técnicas de precarga y reutilización es de 40 ciclos de reloj. Ya que para este tamaño la frecuencia de reloj es de 100 MHz, el retardo es sólo de 400 ns, que es despreciable, especialmente si tenemos en cuenta que en este caso nuestro gestor ha escondido las latencias de reconfiguración de tres subtareas y previene una costosa reconfiguración mediante la reutilización de una subtarea.

Tabla 5. Retardos introducidos por el gestor

Evento	Nuevo grafo	Fin de rec. (subtarea 1)	Fin de ej. (subtarea 1)	Fin de ej. (subtarea 3)
Ciclos	16	2	11	11

Tabla 6. Coste de implementación para un gestor con una tabla asociativa de ocho entradas y tres URs

Módulo	Número de slices	Slices (%)	Block RAMs	RAMs (%)
Unidad de control	21	0.2%	0	0%
FIFO rec.	8	0.2%	1	0,6%
UR	38	0.2%	1	0,6%
Cola de eventos	8	0.2%	1	0,6%
Tabla asociativa	307	2%	1	0,6%
Controlador DMA	621	4%	0	0%
Gestor	**1117 (496)**	**7% (3%)**	**6**	**2,6%**

La tabla 6 muestra el coste de cada uno de estos módulos y el coste total del gestor de ejecución. Este gestor necesita el 7% de los recursosos de la FPGA pero sólo necesita el 3% si no incluimos el controlador DMA. En este caso el módulo más costoso es la tabla asociativa ya que ha sido diseñada para dar el mayor rendimiento posible. Sin embargo, el coste es todavía bastante reducido para este tamaño.

La tabla asociativa es además el módulo con mayor retardo. Por lo tanto, resulta interesante su implementación para distintos tamaños, con el fin de estudiar la evolución de su coste y su retardo. Estos datos se presentan en la tabla 7. Como se puede ver, el área necesaria crece linealmente con respecto al tamaño de la tabla. Por ejemplo una tabla con 16 entradas utiliza el 4% de los recursos de la FPGA y una tabla de 32 entradas utiliza el 8%. Sin embargo, el periodo de reloj soportado es prácticamente constante, 100 MHz, cuando la tabla incluye la modificación para la operación de inserción descrita anteriormente. Sin embargo, se debe remarcar que las FPGAs actuales incluyen soporte para el uso de múltiples relojes, por lo tanto, la frecuencia de reloj de 100 MHz puede que sólo reduzca el rendimiento de nuestro gestor, pero no el rendimiento de las subtareas que son ejecutadas en las URs, ya que estas pueden utilizar su propio reloj.

Tabla 7. Coste y frecuencia de reloj para distintos tamaños de la tabla asociativa

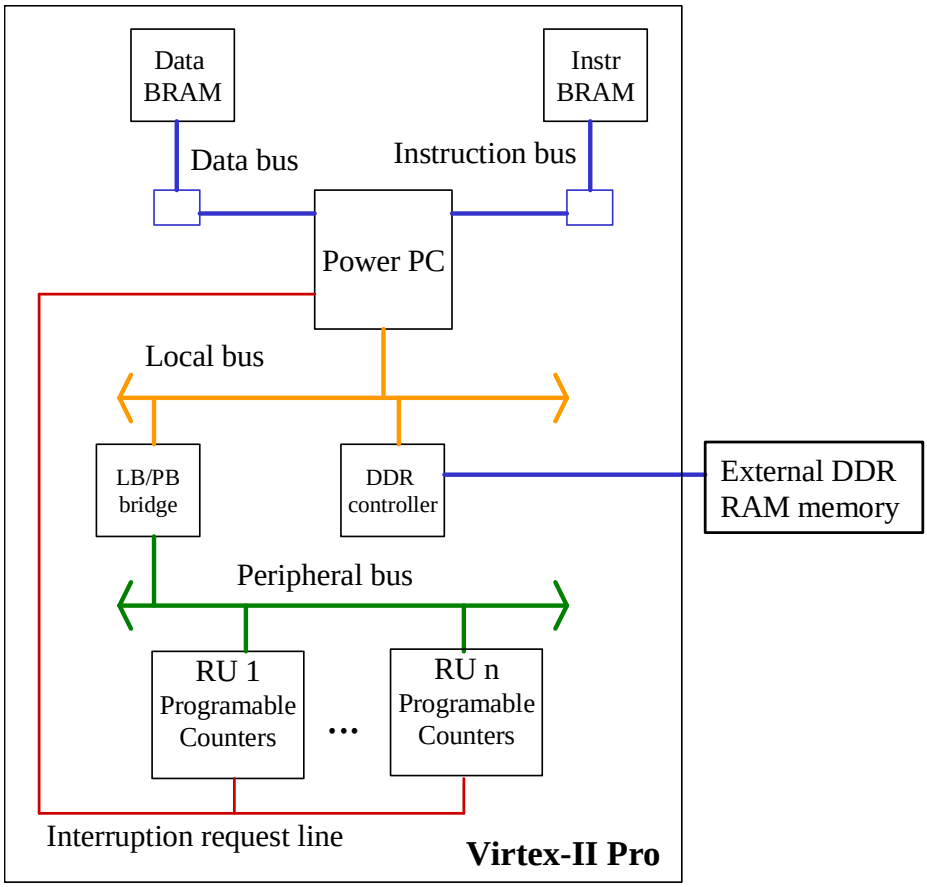
Tamaño	8	16	32	64
Slices (%)	2	4	8	16
Block RAMs(%)	0	0	0	0
Frecuencia de reloj (MHz)	100	100	99,2	98,7

En sistemas empotrados el tamaño del código es también una medida importante debido a que normalmente el espacio de memoria es muy reducido. Por ejemplo, en la FPGA Virtex-II PRO xc2vp30 el tamaño de la RAM interna es de 306KB. El tamaño del código de la implementación SW del gestor es de 190KB mientras que para la versión HW es de 82KB. “Estos tamaños no sólo incluyen el gestor pero todos los drivers para los periféricos incluidos en el sistema”. Por tanto, que a pesar de que la implementación HW del gestor tenga un coste de área HW significante, reduce enormente los requerimientos de memoria del sistema. Esto puede útil ser sobre todo para reducir los accesos a memorias externas.

EVALUACIÓN DEL RENDIMIENTO

En este apartado se va a evaluar el rendimiento de las dos implementaciones del gestor. La versión SW ha sido desarrollada por un compañero del grupo de investigación, por lo que no se van a dar detalles de implementación, aunque una visión general se recoge en la figura 34. Como puede observarse, la principal diferencia es la desaparición del gestor HW (cuya tarea ahora será realizada por el procesador) y la comunicación direccta entre el procesador y las URs a través de las líneas de interrupción (por lo que aumentarán las comunicaciones HW/SW). En esta versión, las URs son capaces de generar interrupciones para avisar al procesador

Figura 34. Implementación SW equivalente



Ambas se han implementado en una FPGA Virtex-II PRO xc2vp30 utilizando el entorno EDK, y se han evaluado utilizando un conjunto de grafos de tareas sacados de aplicaciones multimedia actuales, incluyéndose dos versiones de un decodificador JPEG (JPEG y Parallel-JPEG), un codificador MPEG-1, un reconocedor de patrones (HOUGH), y una aplicación de renderización 3D basada en la librería de código libre Pocket-GL [62] (Pocket GL). En el último caso la aplicación incluye 20 grafos diferentes, por eso la tabla sólo presenta la media de los resultados. Estos grafos han sido planificados para una plataforma con 4 URs utilizando el entorno de planificación presentado en [63], sin embargo, se puede utlizar cualquier otra planificación.

La tabla 8 presenta los retardos que el gestor introduce para estos grafos. Las columnas 2 y 3 incluyen el número de subtareas de cada grafo dirigido acíclico y el tiempo de ejecución inicial asumiendo que la gestión de la ejecución no genera ninguna penalización. Después, las columnas de la 4 a la 7 proporcionan detalles en cuanto a la penalización generada por la versión SW, mientras que la última columna se refiere a la versión HW. Estos retardos se deben al tiempo de computación y a las comunicaciones HW/SW entre el procesador y las URs.

Tabla 8. Evaluación del rendimiento del gestor de ejecución de grafos de tareas

Tarea	Número de subtareas	Tiempo inicial de ejecución (ms)	RENDIMIENTO DEL GESTOR				
			SW				HW
			Tiempo de ejecución (ms)	Tiempo de gestión (ms)		% penaliza-ción	Tiempo de ejecución (ms)
				Cómputos	Comuni-caciones		
JPEG	4	83	83.87	0.466	0.405	1.04	83.022
PARALLEL-JPEG	8	70	71.42	0.609	0.809	1.99	71.023
MPEG-1	5	43	44.02	0.561	0.455	2.31	43.023
HOUGH	6	98	98.88	0.323	0.557	0.89	98.022
POCKET GL	5	44	45.02	0.511	0.510	2.85	44.022

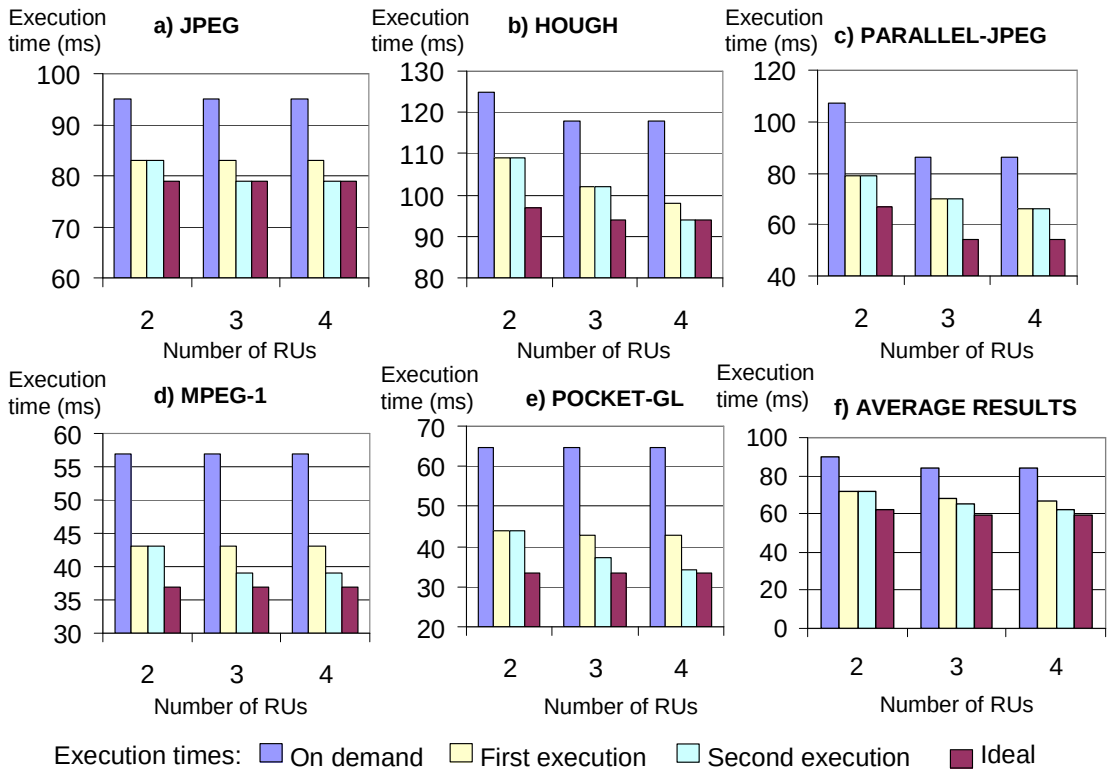
Para la implementación SW, el tiempo de computación inluye el tiempo de ejecución del código que trata sobre la gestión del grafo de tareas y el tiempo de las comunicaciones inluyendo las líneas de interrupción (que en cada cambio implica cambiar al correspondiente ISR, parar/continuar el contador y habilitar/deshabilitar esa interrupción) y la inicialización de las URs cada vez que una nueva subtarea se asigna a ellas. La columna 4 muestra el tiempo actual de ejecución y las columnas 5 y 6 muestran los retardos introducidos por la implementación SW del gestor en tiempos de cómputo y comunicación. Finalmente, la columna 7 muestra el impacto que estos retardos tienen sobre el funcionamiento general de las aplicaciones.

Para la implementación HW, el tiempo de gestión incluye ambos, el retardo introducido por la lógica del gestor y el tiempo de las comunicaciones. La columna 8 muestra el retardo total producido por esta versión. En este caso, el retardo introducido debido a la computación para la gestión son solamente unos pocos cientos de ciclos en un sistema funcionando a 100 MHz (p. ej. unos pocos milisegundos). Respecto a las comunicaciones, estos retardos son de aproximadamente 2200 ciclos cuando se utiliza un DMA.

Como puede verse en la tabla, la implementación SW genera pequeños retardos que van desde el 1% al 3% respecto al tiempo inicial de ejecución. Si estos retardos no son aceptables, se pueden reducir enormemente los tiempos de computación y ejecución utilizando la implementación HW. Así, el tiempo de comunicación se reduce de un promedio de 0,54 ms a tan sólo 0,022 ms, y el tiempo de gestión se reduce drásticamente (al menos en tres órdenes de magnitud).

La figura 35 muestra la reducción de la penalización de la reconfiguración debido a las técnicas optimizadas aplicadas al gestor. La figura incluye cuatro tiempos de ejecución. *On demand* presenta una situciación en la que no se realiza ninguna optimización. Las reconfiguraciones comienzan cuando una subtarea puede ser ejecutada. *First execution* representa la misma ejecución pero aplicando la optimización de prebúsqueda. *Second execution* muestra los beneficios de la técnica de reutilización si un grafo se ejecuta dos veces consecutivas. En esta situación alunas veces es posible reutilizar alguna de las subtareas, lo que reduce enormemente las penalizaciones por reconfiguración. Finalmente *Ideal* representa el tiempo de ejeciución sin penalizaciones por reconfiguación.

Figura 35. Impacto de la optimización de la prebúsqueda y reutilización en el funcionamiento del sistema



En estos experimentos la latencia de reconfiguración se ha establecido en 4 ms, que es el tiempo necesario para reconfigurar un quinto de la FPGA XC2VP30, ya que todas las subtareas consideradas en este experimento caben en esa área. Los resultado muestran que en *on demand* es muy ineficiente y que las optimizaciones de prebúsqueda pueden pueden mejorar enormemente el funcionamiento del sistema. Por regla general las reconfiguraciones generan una penalización del 42% en tiempo de ejecución cuando se utiliza la aproximación *on demand*, sin embargo sólo se genera alrededor de un 13% de penalización cuando se utiliza la técnica de prebúsqueda. Además, esta penalización se reduce a tan sólo el 9% cuando el gestor puede reutlizar alguna de las subtareas. En este experimento se ha utlizado un pequeño número de URs para crear un escenario complejo. Sin embargo, si el tiempo de reconfiguración es todavía elevado, muestro gestor puede reducir las penalizaciones si se incluyen más URs. Por ejemplo, la figura 35 no muestra ningún beneficio por la reutilización de tareas para la aplicación Parallel-JPEG, pero si se incluye una UR adicional (p. ej. cinco URs en total) una de las subtareas puede ser reutilizada y las latencias debidas a las reconfiguraciones se reducirían a 4 ms.

Es importante mencionar que en la computación se consideran las técnicas de prebúsqueda y reutilización para los resultados mostrados anteriormente en la tabla 8. Por ello, aunque el gestor aplique estas técnicas en tiempo de ejecución, no genera apenas retardos en tiempo de ejecución. Esto es especialmente verdadero en el caso de la versión HW del gestor. Esto es una mejora importante cuando comparamos con otras técnicas de prebúsqueda presentadas anteriormente, porque, como se ha explicado en el apartado de trabajo relacionado, la mayoría de ellas no pueden aplicarse en tiempo de ejecución, mientras otras proponen cómputos complejos en tiempo de ejecución, pero no evaluan la penalización en tiempo de ejecución generada por esos cómputos.

CONCLUSIONES

En esta memoria se ha presentado un gestor de ejecución HW para sistemas multitareas. Este gestor recibe como entrada grafos de tareas planificados, y garantiza su correcta ejecución teniendo en cuenta las dependencias entre subtareas. Su objetivo es mejorar la eficiencia de la gestión de tareas en sistemas multiprocesador reconfigurables, reduciendo las costosas comunicaciones HW/SW e incluyendo soporte HW para tratar operaciones con tipos de datos complejos rápidamente.

También, se ha comparado el coste y el rendimiento entre la versión HW del gestor y una versión SW equivalente para un conjunto de aplicaciones multimedia actuales. Respecto al rendimiento, la versión SW introduce retardo en tiempo de ejecución que va desde el 1% al 3% del tiempo total de ejecución, mientras que la versión HW sólo genera retardos insignificantes. La implementación HW introduce una penalización extra en área (casi el 7% de los recursos HW disponibles) pero se reduce enormemente el tamaño del código del procesador empotrado.

Además, se han comprobado los beneficios de la utilización de las técnicas de precarga y reutilización. Los resultados demuestran que estas técnicas pueden ser muy efectivas para reducir la latencia de reconfiguración. En los experimentos realizados han reducido la penalización inicial del 42% a tan sólo el 9%.

PUBLICACIONES GENERADAS

Se presentan a continuación las publicaciones mediante las que se ha divulgado el trabajo de investigación realizado en esta memoria. Están ordenadas siguiendo un criterio que sitúa inicialmente las publicaciones donde se presentan los avances del gestor; y finalmente la publicaciós relacionada con la planificación de grafos de tareas:

- **"HW implementation of an execution manager for reconfigurable systems"**, Javier Resano, Juan Antonio Clemente, Carlos Gonzalez, Jose Luis Garcia and Daniel Mozos, Engineering of Reconfigurable Systems and Algorithms (ERSA), 2007.

- **"Un sistema para la gestión eficiente del HW reconfigurable",** Carlos González, Juan Antonio Clemente, José Luis García, Javier Resano y Daniel Mozos, Jornadas sobre Computación Reconfigurable y Aplicaciones (JCRA), 2007.

- **"Task-graph management for reconfigurable multi-tasking systems"**. Juan Antonio Clemente, Carlos González, Javier Resano and Daniel Mozos, Proceedings of ReCoSoc'2008. Barcelona, July 9-11.

- **"Efficiently scheduling run-time reconfigurations"** Javier Resano, Juan Antonio Clemente, Carlos González, Daniel Mozos, Francky Catthoor. Transactions on Design Automation of Electronic Systems, 2008.

TRABAJO FUTURO

Podemos considerar que prácticamente todos los objetivos propuestos al inicio del desarrollo del proyecto han llegado a cumplirse: se ha diseñado, implementado y testeado un gestor para la ejecución de tareas HW, así como una comparativa con una versión SW equivalente del sistema. De este modo se ha podido comprobar que es una buena elección optar por el HW dinámicamente reconfigurable para ejecutar tareas dadas como grafos de subtareas, y que las operaciones de gestión que se realizan en dichos grafos se ejecutan mucho más rápidamente en su versión HW que en su versión SW (tabla 8).

Es evidente que todo diseño HW se puede mejorar: se puede hacer que trabaje más rápido (a menor frecuencia de reloj), más eficiente (que realice las operaciones en menos ciclos) y que, al ser sintetizado en una FPGA, consuma menos recursos (en términos de área). El gestor propuesto no es una excepción. En concreto, queremos diseñar una tabla asociativa más escalable, y también reducir su retardo mediante la aplicación de técnicas de anticipación de operandos en la red iterativa y/o añadiendo algún ciclo extra. Esto haría que la frecuencia mínima de reloj a la que el sistema funcionase correctamente se redujera, con la consecuente ganancia en velocidad.

Además, se pretende implementar un sistema HW multi-tarea basado en reconfiguración parcial, por lo que será necesario investigar sobre como poderla llevar acabo de manera eficiente.

Por último, y como trabajo a medio-largo plazo, se puede diseñar un planificador de tareas que interactúe con nuestro gestor. De este modo, tendríamos un sistema completo que recibiría el grafo de una tarea a ejecutar, ésta sería analizada en tiempo de diseño y/o tiempo de ejecución (según los algoritmos utilizados) y el gestor recibiría esta planificación para llevarla a cabo. En este sentido, hay trabajos relacionados en los que se plantea la posibilidad de crear un planificador de tareas que se ejecutan en distintas unidades reconfigurables utilizando técnicas de precarga. Asimismo, se plantea la posibilidad de aplicar técnicas de minimización de costes de reconfiguración en sistemas dinámicamente reconfigurables a una planificación dada, con lo que se podrían incorporar esas ideas. Todas estas ampliaciones ayudarían, sin duda, a proporcionar soporte a la investigación del HW reconfigurable en general y al desarrollo de una tecnología con posibles usos comerciales en el futuro; algo que nos parece especialmente interesante.

GLOSARIO DE TÉRMINOS

Asíncrono:
Hace referencia al suceso que no tiene lugar en total correspondencia temporal con otro suceso. Si se refiere a una comunicación asíncrona, ésta es en la que se realiza el envío de datos sin la sincronización de un reloj externo.

Bit:
Dígito en el sistema binario.

Bus:
Conjunto de conductores eléctricos en forma de pistas metálicas impresas sobre la placa base del computador, por donde circulan las señales que corresponden a los datos binarios del lenguaje máquina. Si el bus es de datos, este conectará dos dispositivos hardware.

Byte:
8 bits.

Chip:
Circuito integrado o pastilla en la que se encuentran todos o casi todos los componentes necesarios para que un ordenador pueda realizar alguna función.

Ciclo:
Un ciclo es la distancia temporal entre el principio y el final de una onda completa.

Circuito impreso:
Medio para sostener mecánicamente y conectar eléctricamente componentes electrónicos, a través de rutas o pistas de material conductor, grabados desde hojas de cobre laminadas sobre un sustrato no conductor.

Depuración:
Proceso de mejora de un sistema hasta que realiza su función correctamente.

Empotrado:
Hubicado dentro del chip.

Esquemático:
Es un diagrama, dibujo o boceto que detalla los elementos de un sistema.

Evento:
Acontecimiento ocurrido en el sistema.

FIFO:
First In, First Out. Protocolo que implementa una cola: lo que viene primero, se maneja primero, lo que viene segundo espera hasta que lo primero haya sido manejado, etc.

Flanco:
Transición del nivel bajo al alto (flanco de subida) o del nivel alto al bajo (flanco de bajada) de una señal.

FPGA:
Dispositivo donde se pueden programar infinidad de diseños digitales distintos.

Gestor:
Elemento que se encarga de organizar y dar órdenes a otros elementos de menor nivel en la jerarquía de diseño.

Grafo:
Conjunto de nodos relacionados mediante aristas de simple o doble sentido.

Hardware:
Es un término general usado para describir artefactos físicos de una tecnología.

Planificación:
Modelo de ejecución para una tarea.

Precarga:
Técnia que preve y lleva a cabo una reconfiguración parcial con anterioridad a su utilización.

Predecesor:
Que se encuentra con anterioridad en la jerarquía.

Prototipo:
Ejemplar original o primer molde en que se fabrica un diseño.

Reconfiguración:
Proceso en el que parte de la FPGA (reconfiguración parcial) o toda (reconfiguración total) se reprograma.

Síncrono:
Hace referencia al suceso que tiene lugar en correspondencia temporal con otro suceso. Si se refiere a una comunicación síncrona, ésta es en la que se realiza el envío de datos bajo la sincronización de un reloj externo.

Speed-up:
Cociente que nos dice la mejora de rendimiento obtenida.

Subtarea:
Cada una de las partes de una tarea con significado propio.

Sucesor:
Que se encuentra con posterioridad en la jerarquía.

Unidad reconfigurable:
Parte de la FPGA que puede ser reconfigurada de modo independiente.

VHDL:
Very High Speed Integrate Circuit Hardware Description Language, es un lenguaje de descripción y modelado, diseñado para describir la funcionalidad y la organización de sistemas *hardware* digitales, placas de circuitos, y componentes.

BIBLIOGRAFÍA

[1] P. Lysaght et al.," Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs", FPL'06, p. 1-6, 2006.

[2] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, R. Lauwereins, "Interconnection Network enable Fine-Grain Dynamic Multi-Tasking on FPGAs", FPL,p. 795-805, 2002.

[3] www.xilinx.com/products/silicon_solutions/index.htm. May, 2008.

[4] www.altera.com/products/devices/stratix-fpgas/about/stx-about.html. May, 2008.

[5] www.xilinx.com/ise/embedded/edk_docs.htm. May, 2008.

[6] www.altera.com/products/software/products/sopc/sop-index.html. May, 2008.

[7] P. Garcia, K. Compton, M. Schulte, E. Blem and W. Fu. "An Overview of Reconfigurable Hardware in Embedded Systems". EURASIP Journal on Embedded Systems, vol. 2006.

[8] G. E. Moore: *Cramming More Components onto Integrated Circuits*. Electronics, 38(8), Abril 1965.

[9] L. Waller: *The Big Question in Counting FPGA Gates: Should Memory Be Included*. EE Times Online, http://www.eedesign.com/editorial/1997/fpgacolumn9710. html, Octubre 1997.

[10] Y.W. Chan: *Physical design for SoC*. Tutorial Soc pd2http://http://cc.ee.ntu. edu.tw/~ywchang, 2002.

[11] R. Rajsuman: *System-on-a-Chip. Design and Test.* Artech House Publishers, 2000.

[12] J. Borel: *SoC Design Challenges: the EDA MEDEA Roadmap.* Servicio de Publicaciones de la Universidad de Cantabria, 2001.

[13] A. Chao: *Platform Design System SoC*. Faraday Technology Corporation, http: //www.faraday-tech.com, 2002.

[14] Virtual Socket Interface Aliance: *Speci¯ cations, Standars, Technical Documents* http://www.vsia.org, 2003.

[15] G. Martin y H. Chang (Eds.): *Winning the SoC Revolution: Experiences in Real Design*. Kluwer Academic Publishers, Massachusetts, USA, 2003.

[16] www.xilinx.com/ise/embedded/edk_docs.htm. May, 2008.

[17] www.altera.com/products/software/products/sopc/sop-index.html. May, 2008.

[18] P. Garcia, K. Compton, M. Schulte, E. Blem and W. Fu. "An Overview of Reconfigurable Hardware in Embedded Systems". EURASIP Journal on Embedded Systems, vol. 2006.

[19] H. Walder, M. Platzner, "Online Scheduling for Block-partitioned Reconfigurable Devices". DATE'03 p. 10290-10295. 2003.

[20] Y. Qu, J. Soininen, J..Nurmi, "A Parallel Configuration Model for Reducing the Run-Time Reconfiguration Overhead", DATE'06, pp. 965- 969. 2006.

[21] J. Noguera, M. Badia., “Multitasking on Reconfigurable Architectures: Microarchitecture Support and Dynamic Scheduling”. ACM Transactions on Embedded Computing Systems, Vol. 3, No. 2, p. 385-406. 2004

[22] J. Javier Resano, M. Elena Pérez, Daniel Mozos, Hortensia Mecha, Julio Septién. “Analyzing Communication Overheads during Hardware/Software Partitioning”. Elsevier Microelectronic Journal, vol. 34-11, pag. 1001-1007. 2003.

[23] R. P. Dick, G. Lakshminarayana, A. Raghunathan, and N.K. Jha. “Power analysis of embedded operating systems”. In Proceeding Design Automation Conference (DAC),2000.

[24] B.E. Saglam, V. Andmooney. “System-on-a-chip processor synchronization support in hardware”. In Proceedings of Design Automation and Test in Europe (DATE’01). 2001.

[25] K. Compton y S. Hauck: *Reconfigurable Computing: A Survey of Systems and Software*. ACM Computing Surveys, 34(2):171-210, Junio 2002.

[26] J. Tuley: *Soft Computing Reconfigures Designer Options*. Embedded Systems, página 76, Abril 1997.

[27] K. Bondalapati y V.K. Prasanna: *Reconfigurable Computing Systems*. Proceedings of the IEEE, 90(7):1201-1217, Julio 2002.

[28] A. DeHon: *Reconfigurable Architectures for General Purpose Computing*. A.I. Technical report No. 1586. Artificial Intelligence Laboratory. Massachusetts Institute of Technology, Septiembre 1996.

[29] R. Tessier y W. Burleson: *Reconfigurable Computing for Digital Signal Processing: a survey*. Journal of VLSI Signal Processing, (28):7-27, Mayo 2001.

[30] R. Sidhu, A. Mei y V.K. Prassana: *Genetic Programming using Self-Reconfigurable FPGAs*. Lecture Notes in Computer Science, 1673:301-312, 1999.

[31] R. Sidhu y A. Mei V.K. Prasanna: *String Matching on Multicontext FPGAs using Self-Reconfiguration*. En *Proceedigns of the ACM International Symposium in Field Programmable Gate Arrays (FPGA'99)*, páginas 217-226, Febrero 1999.

[32] R. Sidhu y V.K. Prasanna: *Fast Regular Expression Matching using FPGAs*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, páginas 698-709, Abril 2001.

[33] J. Hauser y J. Wawrzynek: *Garp: A MIPS Processor with a Reconfigurable Coprocessor*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'97)*, páginas 12-21, Abril 1997.

[34] K.H. Leung, K.W. Wong y P.H.W. Leong: *FPGA Implementation of a Microcoded Elliptic Curve Cryptographic Processor*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*, páginas 68-76, Abril 2000.

[35] J. Lázaro, A. Astarloa, U. Bidarte, J. Arias y C. Cuadrado: *High Throughput Serpent Encryption Implementation*. Lecture Notes in Computer Science, 3203:996-1000, 2004.

[36] W.J. Huang and N. Saxena y E.J. McCluskey: *A reliable LZ data compressor on reconfigurable coprocesors*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*, páginas 249-258, Abril 2000.

[37] J. Burns, A. Donlin, J. Hogg, S. Singh y M. De Wit: *A Dynamic Reconfiguration Run-Time System*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'97)*, páginas 66-76, Abril 1997.

[38] J. Gause, P.Y.K. Cheung y W. Luk: *Reconfigurable Shape-Adaptive Template Matching Architectures*. En *Proceedigns of the IEEE FPGA Custom Computing Machine Conference 2002*, páginas 98-110, 2002.

[39] M. Ahrens, A. El Gamal, D. Galbraith, J. Greene y S. Kaptanoglu: *An FPGA family optimized for high densities and reduced routing delay*. En *Proceedings of the IEEE Custom Integrated Circuits Conference*, páginas 31.5.1-31.5.4, 1990.

[40] H. Hsieh, W. Carter, J. Y. Ja, E. Cheung, S. Schreifels, C. Erickson, P. Freidin y L. Tinkey: *Third-generation Architecture Boosts Speed and Density of Field-programmable Gate arrays*. En *Proceedings of the IEEE Custom Integrated Circuits Conference*, páginas 31.2.1-31.2.7, 1990.

[41] Actel Corporation: *Accelerator Series FPGAs: ACT3 Family*. http://www.actel.com, 1997.

[42] Actel Corporation: *SX Family of High Performance FPGAs*. http://www.actel.com, 2001.

[43] S. Hauck, T.W. Fry, M.M. Hosler y J.P. Kao: *The Chimaera Reconfigurable Functional Unit*. IEEE Transactions on VLSI Systems, 12(2):206-217, Febrero 2004.

[44] C.R. Rupp, M. Landguth, T. Garverick, E. Comersall, H. Holt, J.M. Arnold y M. Gokhale: *The NAPA adaptive processing architecture*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'98)*, páginas 28-37, Abril 1998.

[45] Xilinx Corp.: *Using Block SelectRAM+ Memory in Spartan II FPGAs*. Xilinx Application Notes, http://www.xilinx.com, Deciembre 2000.

[46] Inc. Chameleon Systems: *CS2000 Advance Product Specification*. http://www.isis.vanderbilt.edu/projects.asp, 2000.

[47] Z. Li, K. Compton y S. Hauck: *Configuration Caching Techniques for FPGA*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*, páginas 22-38, Abril 2000.

[48] T. Anderson: *System-on-Chip Design with Virtual Components*. Circuit Cellar Online, http://www.circuitcellar.com, Agosto 1999.

[49] S. Trimberger, D. Carberry, A. Johnson y J. Wong: *A time-multiplexed FPGA*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'97)*, páginas 22-28, Abril 1997.

[50] A. DeHon: *DPGA Utilization and Applications*. En *Proceedigns of the ACM International Symposium in Field Programmable Gate Arrays (FPGA'96)*, páginas 115-121, Febrero 1996.

[51] K. Danne, C. Bobda y H. Kalte: *Run-Time Exchange of Mechatronic Controllers Using Partial Hardware Reconfiguration*. Lecture Notes in Computer Science, 2778:272-281, 2003.

[52] Y. Chou, P. Pillai, H. Schmit y J.P. Shen: *PipeRench Implementation of the Instruction Path Coprocessor*. En *Proceedigns of the 33th Annual International Symposium on Microarchitecture (MICRO)*, páginas 147-158, 2000.

[53] Xilinx Corp.: *XC6200 Field Programmable Gate Arrays*. Xilinx Documentation, http://www.xilinx.com, 2002.

[54] Xilinx Corp.: *Virtex 2.5V Field Programmable Gate Array*. Xilinx Documentation, http://www.xilinx.com, Deciembre 2002.

[55] H. Schmit: *Incremental Reconfiguration for Pipelined Applications*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'97)*, páginas 16-18, Abril 1997.

[56] S. Hauck: *Configuration Prefetch for Single Context Reconfigurable Coprocessors*. En *Proceedigns of the ACM International Symposium in Field Programmable Gate Arrays (FPGA'98)*, páginas 65-74, Febrero 1998.

[57] S. Hauck, Z. Li y E. Schewabe: *Configuration Compression for the Xilinx XC6200 FPGA*. En *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'98)*, páginas 138-147, Abril 1998.

[58] K. Compton, Z. Li, J. Cooey, S. Knol y S. Hauck: *Configuration Relocation and Defragmentation for Run-Time Reconfigurable Computing*. IEEE Transactions on VLSI Systems, 10(3):209-220, Junio 2002.

[59] Xilinx Corp.: *Virtex FPGA Series Configuration and Readback*. Xilinx Application Notes, http://www.xilinx.com, Julio 2002.

[60] Xilinx Corp.: *Virtex Series Configuration Architecture User Guide*. Xilinx Application Notes, http://www.xilinx.com, Febrero 2003.

[61] Xilinx Corp.: *Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations*. Xilinx Application Notes, http://www.xilinx.com, Mayo 2002.

[62] L. Shang, N.K. Jha, “Hw/Sw Co-synthesis of Low Power Real-Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs”, ASP-DAC’02, pp. 345-360, 2002.

[63] J. Resano, D. Mozos, F Catthoor, "A Hybrid Prefetch Scheduling Heuristic to Minimize at Runtime the Reconfiguration Overhead of Dynamically Reconfigurable HW" DATE05, pp.106-111. 2005

[64] J. Noguera, M. Badia., “Power-Performance Trade-Offs for Reconfigurable Computing”. CODES+ISSS.pp. 116-121. 2004

[65] K. N. Vikram, V. Vasudevan. “Mapping Data-Parallel Tasks Onto Partially Reconfigurable Hybrid Processor Architectures”. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Volume 14, Issue 9, Sept. 2006, pp.1010-1023.

[66] J. Resano, D. Mozos, D Verkest, F Catthoor, "*A Reconfiguration Manager for Dynamically Reconfigurable Hardware*", IEEE Design&Test, Vol. 22, Issue 5, pp. 452-460. 2005.

[67] W. Fu, K. Compton. “An execution environment for reconfigurable computing”. Proc. of Field-Programmable Custom Computing Machines (FCCM), 2005, pp.149-158.

[68] V. Nollet, T. Marescaux, D. Verkest, J-Y. Mignolet, S. Vernalde. “Operating System controlled Networkon-Chip”. Proceedings of the Design Automation Conference (DAC), pag. 256-259, 2004.

[69] Xilinx Inc., Virtex-II Pro and Virtex-II Pro X Platform FPGAs:Complete Data Sheet, Xilinx, San Jose, Calif, USA, 2005.

[70] Z. Li, S. Hauck, “Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation”, FPGA’02, pp. 187-195. 2002

APÉNDICES

FUENTES VHDL DEL PROYECTO

Módulos Básicos

Biestable.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

entity biestable is
   port( clk, rst, ld : in std_logic;
      din : in std_logic;
      dout : out std_logic);
end biestable;

architecture biestableArch of biestable is
   signal cs, ns : std_logic;
   begin

   state:
   process( clk, rst )
      begin
         if (rst='1') then
            cs <= '1';
         elsif (clk'event and clk='1') then
            cs <= ns;
         end if;
   end process;

   next_state:
      ns <= din when ld='1' else cs;

   moore_output:
      dout <= cs;

end biestableArch;
```

Codificador.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity priorityEncoder is
    generic( n : integer := 3 );
    port(
       x : in std_logic_vector( 2**n-1 downto 0 );
       y : out std_logic_vector ( n-1 downto 0 );
       gs : out std_logic );
end priorityEncoder;

architecture priorityEncoderArch of priorityEncoder is
begin
    process ( x )
    begin
       y <= (others=>'0'); gs <= '0';
       for i in x'reverse_range loop
        if x(i)='1' then
            y <= conv_std_logic_vector( i, n );
            gs <= '1';
        end if;
       end loop;
```

```
    end process;
end priorityEncoderArch;
```

Contador.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity contador is
   generic( n : integer := 8 );
   port( clk, rst, ld, dec : in std_logic;
      din : in std_logic_vector( n-1 downto 0 );
      dout : out std_logic_vector( n-1 downto 0 ) );
end contador;

architecture contadorArch of contador is
   signal cs, ns : std_logic_vector( n-1 downto 0 );
   begin

   state:
   process( clk, rst )
     begin
        if rst='1' then cs <= (others=>'0');
        elsif clk'event and clk='1' then cs <= ns;
        end if;
   end process;

   next_state:
   process( cs, ld, dec, din )
      begin
         if ld='1' then ns <= din;
         elsif dec='1' then ns <= cs - 1;
                  -- no es necesario detectar el final de cuenta
         else ns <= cs;
         end if;
   end process;

   moore_output: dout <= cs;

end contadorArch;
```

ContadorInc.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity contadorInc is
   generic( n : integer := 8 );
   port( clk, rst, inc : in std_logic;
      dout : out std_logic_vector( n-1 downto 0 ) );
end contadorInc;

architecture contadorIncArch of contadorInc is
   signal cs, ns : std_logic_vector( n-1 downto 0 );
   begin

   state:
   process( clk, rst )
     begin
        if rst='1' then cs <= (others=>'0');
        elsif clk'event and clk='1' then cs <= ns;
        end if;
   end process;

   next_state:
```

```
    process( cs, inc )
       begin
          if inc='1' then ns <= cs + 1;
          else ns <= cs;
          end if;
    end process;

    moore_output: dout <= cs;

end contadorIncArch;
```

ContadorIncDec.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity contadorIncDec is
   generic( n : integer := 8 );
   port( clk, rst, inc, dec : in std_logic;
      dout : out std_logic_vector( n-1 downto 0 ) );
end contadorIncDec;

architecture contadorIncDecArch of contadorIncDec is
   signal cs, ns : std_logic_vector( n-1 downto 0 );
   begin

   state:
   process( clk, rst )
     begin
        if rst='1' then cs <= (others=>'0');
        elsif clk'event and clk='1' then cs <= ns;
        end if;
   end process;

   next_state:
   process( cs, dec, inc )
      begin
         if inc='1' then ns <= cs + 1; -- prioridad al incremento
         elsif dec='1' then ns <= cs - 1;
         else ns <= cs;
         end if;
   end process;

   moore_output: dout <= cs;

end contadorIncDecArch;
```

ContadorIncDecMismoCiclo.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity contadorIncDecMismoCiclo is
   generic( n : integer := 8 );
   port( clk, rst, inc, dec : in std_logic;
      dout : out std_logic_vector( n-1 downto 0 ) );
end contadorIncDecMismoCiclo;

architecture contadorIncDecMismoCicloArch of contadorIncDecMismoCiclo is
   signal cs, ns : std_logic_vector( n-1 downto 0 );
   begin

   state:
```

```
    process( clk, rst )
      begin
         if rst='1' then cs <= (others=>'0');
         elsif clk'event and clk='1' then cs <= ns;
         end if;
    end process;

    next_state:
    process( cs, dec, inc )
       begin
          if inc='1' and dec='1' then
             ns <= cs;
          elsif inc='1' then
       ns <= cs + 1;
          elsif dec='1' then
       ns <= cs - 1;
          else
       ns <= cs;
          end if;
    end process;

    moore_output: dout <= cs;

end contadorIncDecMismoCicloArch;
```

Decodificador.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity decoder is
    generic( n : integer := 3 );
    port(
       x : in std_logic_vector( n-1 downto 0 );
       en : in std_logic;
       y : out std_logic_vector ( 2**n-1 downto 0 ) );
end decoder;

architecture decoderArch of decoder is
begin
    process ( x, en )
    begin
       y <= (others=>'0');
       if en='1' then
          y(conv_integer(unsigned(x))) <= '1';
       end if;
    end process;
end decoderArch;
```

Registro.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity registro is
   generic( n : integer := 8);
   port( clk, rst, ld : in std_logic;
      din : in std_logic_vector( n-1 downto 0 );
      dout : out std_logic_vector( n-1 downto 0 ) );
end registro;

architecture registroArch of registro is
   signal cs, ns : std_logic_vector( n-1 downto 0 );
   begin
```

```
    state:
    process( clk, rst )
       begin
          if (rst='1') then
             cs <= (others=>'0');
          elsif (clk'event and clk='1') then
             cs <= ns;
          end if;
    end process;

    next_state:
       ns <= din when ld='1' else cs;

    moore_output:
       dout <= cs;

end registroArch;
```

FIFOs

Fifo3Bits.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;


entity fifo3bits is
   generic (numentradaslog : integer := 3 );
   port (entradadatos: in std_logic_vector(2 downto 0);
     clk, rst, leer, escribir, enable : in std_logic;
          salidadatos: out std_logic_vector(2 downto 0);
     fifollena, fifovacia: out std_logic);
end fifo3bits;

architecture fifo3bitsarch of fifo3bits is

component ramb16_s4
-- AAAsynthesis translate_off
generic (
       srval : bit_vector := x"0";
       write_mode : string := "no_change");
   -- AAAsynthesis translate_on
   port (do : out std_logic_vector (3 downto 0);
         addr : in std_logic_vector (11 downto 0);
         clk : in std_ulogic;
         di : in std_logic_vector (3 downto 0);
         en : in std_ulogic;
         ssr : in std_ulogic;
         we : in std_ulogic);
end component;

   component contadorinc
      generic( n : integer := 9 );
      port( clk, rst, inc : in std_logic;
            dout : out std_logic_vector( n-1 downto 0 ) );
   end component;

  component contadorincdecmismociclo
      generic( n : integer := 8 );
      port( clk, rst, inc, dec : in std_logic;
         dout : out std_logic_vector( n-1 downto 0 ) );
   end component;
```

```
    signal salidacontadorprimero,
      salidacontadorultimo :std_logic_vector(numentradaslog-1 downto 0);
    signal salidacontadornumocupadas,
      ceros, unos :std_logic_vector(numentradaslog downto 0);
    signal incprimero, incultimo, escribirleer, enablefifo :std_logic;
    signal direccion: std_logic_vector(11 downto 0);
    signal entradafifo, salidafifo : std_logic_vector(3 downto 0);

begin

ram : ramb16_s4
    -- AAAsynthesis translate_off
   -- AAAsynopsys translate_on
port map (do => salidafifo,
              addr => direccion,
              clk => clk,
              di => entradafifo,
              en => enablefifo,
              ssr => rst,
              we => escribirleer);

    enablefifo <= leer or escribir;
    escribirleer <= '1' when (escribir='1') else '0';
    direccion(11 downto numentradaslog) <= (others=> '0');
    direccion (numentradaslog-1 downto 0) <=
           salidacontadorultimo when (escribir='1') else
           salidacontadorprimero;

    entradafifo (3) <= '0';
    entradafifo (2 downto 0) <= entradadatos;
    salidadatos <= salidafifo (2 downto 0);

    ceros <= (others =>'0');
    unos <= conv_std_logic_vector(2**numentradaslog,numentradaslog+1);

    contadorprimero: contadorinc generic map(numentradaslog)
           port map(clk, rst, incprimero, salidacontadorprimero);
    contadorultimo: contadorinc generic map(numentradaslog)
           port map(clk, rst, incultimo, salidacontadorultimo);

    incprimero <= '1' when (escribirleer ='0'  and enablefifo ='1') else '0';
    incultimo <= '1' when (escribirleer ='1'  and enablefifo ='1') else '0';

    contadornumocupadas: contadorincdecmismociclo
           generic map(numentradaslog+1)
           port map(clk, rst, incultimo, incprimero,
                                          salidacontadornumocupadas);

    fifollena <= '1' when (salidacontadornumocupadas = unos) else '0';
    fifovacia <= '1' when (salidacontadornumocupadas = ceros) else '0';

end fifo3bitsarch;
```

Fifo5Bits.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity fifo5bits is
   generic (numentradaslog : integer := 3 );
   port (entradadatos: in std_logic_vector(4 downto 0);
     clk, rst, leer, escribir, enable : in std_logic;
     salidadatos: out std_logic_vector(4 downto 0);
     fifollena, fifovacia: out std_logic);
end fifo5bits;
```

```
architecture fifo5bitsarch of fifo5bits is

component ramb16_s9
-- AAAsynthesis translate_off
generic (
       srval : bit_vector := x"0";
       write_mode : string := "no_change"
);
-- AAAsynthesis translate_on
   port (do : out std_logic_vector (7 downto 0);
         dop : out std_logic_vector (0 downto 0);
         addr : in std_logic_vector (10 downto 0);
         clk : in std_ulogic;
         di : in std_logic_vector (7 downto 0);
         dip : in std_logic_vector (0 downto 0);
         en : in std_ulogic;
         ssr : in std_ulogic;
         we : in std_ulogic);
end component;

   component contadorinc
      generic( n : integer := 9 );
      port( clk, rst, inc : in std_logic;
            dout : out std_logic_vector( n-1 downto 0 ) );
   end component;

   component contadorincdecmismociclo
      generic( n : integer := 8 );
      port( clk, rst, inc, dec : in std_logic;
              dout : out std_logic_vector( n-1 downto 0 ) );
   end component;

   signal salidacontadorprimero,
        salidacontadorultimo :std_logic_vector(numentradaslog-1 downto 0);
   signal salidacontadornumocupadas,
        ceros, unos :std_logic_vector(numentradaslog downto 0);
   signal incprimero, incultimo, escribirleer, enablefifo :std_logic;
   signal direccion: std_logic_vector(10 downto 0);
   signal entradafifo, salidafifo : std_logic_vector(7 downto 0);
   signal dip : std_logic_vector (0 downto 0);

begin

ram : ramb16_s9
    -- AAAsynthesis translate_off
   -- AAAsynopsys translate_on
port map (do => salidafifo,
                dop => open,
                addr => direccion,
                clk => clk,
                di => entradafifo,
                dip => dip,
                en => enablefifo,
                ssr => rst,
                we => escribirleer);

    dip <= (others=> '0');

    enablefifo <= leer or escribir;
    escribirleer <= '1' when (escribir='1') else '0';
    direccion(10 downto numentradaslog) <= (others=> '0');
    direccion (numentradaslog-1 downto 0) <=
        salidacontadorultimo when (escribir='1') else salidacontadorprimero;

    entradafifo (7 downto 5) <= (others=>'0');
    entradafifo (4 downto 0) <= entradadatos;
    salidadatos <= salidafifo (4 downto 0);

    ceros <= (others =>'0');
```

```
    unos <= conv_std_logic_vector(2**numentradaslog,numentradaslog+1);

    contadorprimero: contadorinc generic map(numentradaslog)
           port map(clk, rst, incprimero, salidacontadorprimero);
    contadorultimo: contadorinc generic map(numentradaslog)
           port map(clk, rst, incultimo, salidacontadorultimo);

    incprimero <= '1' when (escribirleer ='0'  and enablefifo ='1') else '0';
    incultimo <= '1' when (escribirleer ='1'  and enablefifo ='1') else '0';

    contadornumocupadas: contadorincdecmismociclo
           generic map(numentradaslog+1)
           port map(clk, rst, incultimo, incprimero,
                                          salidacontadornumocupadas);

    fifollena <= '1' when (salidacontadornumocupadas = unos) else '0';
    fifovacia <= '1' when (salidacontadornumocupadas = ceros) else '0';

end fifo5bitsarch;
```

Fifo16Bits.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity fifo16bits is
   generic (numentradaslog : integer := 3 );
   port (entradadatos: in std_logic_vector(15 downto 0);
     clk, rst, leer, escribir, enable : in std_logic;
     salidadatos: out std_logic_vector(15 downto 0);
     fifollena, fifovacia: out std_logic);
end fifo16bits;

architecture fifo16bitsarch of fifo16bits is

component ramb16_s18
-- aaasynthesis translate_off
generic (
       srval : bit_vector := x"0";
       write_mode : string := "no_change"
);
-- AAAsynthesis translate_on
   port (do : out std_logic_vector (15 downto 0);
         dop : out std_logic_vector (1 downto 0);
         addr : in std_logic_vector (9 downto 0);
         clk : in std_ulogic;
         di : in std_logic_vector (15 downto 0);
         dip : in std_logic_vector (1 downto 0);
         en : in std_ulogic;
         ssr : in std_ulogic;
         we : in std_ulogic);
end component;

   component contadorinc
      generic( n : integer := 9 );
      port( clk, rst, inc : in std_logic;
            dout : out std_logic_vector( n-1 downto 0 ) );
   end component;

   component contadorincdecmismociclo
      generic( n : integer := 8 );
      port( clk, rst, inc, dec : in std_logic;
         dout : out std_logic_vector( n-1 downto 0 ) );
   end component;

   signal salidacontadorprimero,
```

```
       salidacontadorultimo :std_logic_vector(numentradaslog-1 downto 0);
    signal salidacontadornumocupadas,
      ceros, unos :std_logic_vector(numentradaslog downto 0);
    signal incprimero, incultimo, escribirleer, enablefifo :std_logic;
    signal direccion: std_logic_vector(9 downto 0);
    signal entradafifo, salidafifo : std_logic_vector(15 downto 0);
    signal dip: std_logic_vector(1 downto 0);

begin

ramb16_s36_instance_name5 : ramb16_s18
port map (do => salidafifo,
              dop => open,
              addr => direccion,
              clk => clk,
              di => entradafifo,
              dip => dip,
              en => enablefifo,
              ssr => rst,
              we => escribirleer);

     dip <= (others=>'0');

    enablefifo <= leer or escribir;
    escribirleer <= '1' when (escribir='1') else '0';
    direccion(9 downto numentradaslog) <= (others=> '0');
    direccion (numentradaslog-1 downto 0) <=
      salidacontadorultimo when (escribir='1') else salidacontadorprimero;

    entradafifo (15 downto 0) <= entradadatos;
    salidadatos <= salidafifo;

    ceros <= (others =>'0');
    unos <= conv_std_logic_vector(2**numentradaslog,numentradaslog+1);

    contadorprimero: contadorinc generic map(numentradaslog)
         port map(clk, rst, incprimero, salidacontadorprimero);
    contadorultimo: contadorinc generic map(numentradaslog)
         port map(clk, rst, incultimo, salidacontadorultimo);

    incprimero <= '1' when (escribirleer ='0'  and enablefifo ='1') else '0';
    incultimo <= '1' when (escribirleer ='1'  and enablefifo ='1') else '0';

    contadornumocupadas: contadorincdecmismociclo
         generic map(numentradaslog+1)
         port map(clk, rst, incultimo, incprimero, salidacontadornumocupadas);

    fifollena <= '1' when (salidacontadornumocupadas = unos) else '0';
    fifovacia <= '1' when (salidacontadornumocupadas = ceros) else '0';

end fifo16bitsarch;
```

Fifo30Bits.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity fifo30bits is
   generic (numentradaslog : integer := 3 );
   port (entradadatos: in std_logic_vector(29 downto 0);
     clk, rst, leer, escribir, enable : in std_logic;
     salidadatos: out std_logic_vector(29 downto 0);
     fifollena, fifovacia: out std_logic);
end fifo30bits;

architecture fifo30bitsarch of fifo30bits is
```

```
component ramb16_s36
-- AAAsynthesis translate_off
generic (
       srval : bit_vector := x"0";
       write_mode : string := "no_change"
);
-- AAAsynthesis translate_on
   port (do : out std_logic_vector (31 downto 0);
         dop : out std_logic_vector (3 downto 0);
         addr : in std_logic_vector (8 downto 0);
         clk : in std_ulogic;
         di : in std_logic_vector (31 downto 0);
         dip : in std_logic_vector (3 downto 0);
         en : in std_ulogic;
         ssr : in std_ulogic;
         we : in std_ulogic);
end component;

   component contadorinc
      generic( n : integer := 9 );
      port( clk, rst, inc : in std_logic;
            dout : out std_logic_vector( n-1 downto 0 ) );
   end component;

   component contadorincdecmismociclo
      generic( n : integer := 8 );
      port( clk, rst, inc, dec : in std_logic;
         dout : out std_logic_vector( n-1 downto 0 ) );
   end component;

   signal salidacontadorprimero,
       salidacontadorultimo :std_logic_vector(numentradaslog-1 downto 0);
   signal salidacontadornumocupadas,
       ceros, unos :std_logic_vector(numentradaslog downto 0);
   signal incprimero, incultimo, escribirleer, enablefifo :std_logic;
   signal direccion: std_logic_vector(8 downto 0);
   signal entradafifo, salidafifo : std_logic_vector(31 downto 0);
   signal dip: std_logic_vector(3 downto 0);

begin

ramb16_s36_instance_name7 : ramb16_s36
port map (do => salidafifo,
             dop => open,
             addr => direccion,
             clk => clk,
             di => entradafifo,
             dip => dip,
             en => enablefifo,
             ssr => rst,
             we => escribir);

    dip <= (others=>'0');

    enablefifo <= leer or escribir;
    escribirleer <= '1' when (escribir='1') else '0';
    direccion(8 downto numentradaslog) <= (others=> '0');
    direccion (numentradaslog-1 downto 0) <=
     salidacontadorultimo when (escribir='1') else salidacontadorprimero;


    entradafifo (31 downto 30) <= (others=>'0');
    entradafifo (29 downto 0) <= entradadatos;
    salidadatos <= salidafifo (29 downto 0);

    ceros <= (others =>'0');
    unos <= conv_std_logic_vector(2**numentradaslog,numentradaslog+1);

    contadorprimero: contadorinc generic map(numentradaslog)
```

```
        port map(clk, rst, incprimero, salidacontadorprimero);
    contadorultimo: contadorinc generic map(numentradaslog)
        port map(clk, rst, incultimo, salidacontadorultimo);

    incprimero <= '1' when (escribirleer ='0'  and enablefifo ='1') else '0';
    incultimo <= '1' when (escribirleer ='1'  and enablefifo ='1') else '0';

    contadornumocupadas: contadorincdecmismociclo
        generic map(numentradaslog+1)
        port map(clk, rst, incultimo, incprimero, salidacontadornumocupadas);

    fifollena <= '1' when (salidacontadornumocupadas = unos) else '0';
    fifovacia <= '1' when (salidacontadornumocupadas = ceros) else '0';

end fifo30bitsarch;
```

Módulos compuestos

UR.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ur is
--nº de bits para decir cuánto tiempo tarda en ejecutar y cargar. suponemos
que es lo mismo
    generic (numbitsetiqueta:natural:=3;
                longitud:natural:=16;
                numentradaslog:natural:=3;
                long_eje_carga:natural:=5);
    port ( clk: in std_logic;
           rst: in std_logic;
           enable: in std_logic;
           entrada_fifo:in std_logic_vector(longitud-1 downto 0);
           escribir: in std_logic;
           comenzar_carga : in std_logic;
           comenzar_ejecucion : in std_logic;
           pedir_carga : out std_logic;
           pedir_ejecucion : out std_logic;
           sub_actual : out std_logic_vector(longitud-1 downto 0);
           tabla_llena: out std_logic;
           estado: out std_logic_vector(2 downto 0);
           vacia: out std_logic;
           escribirenfifoeventos: out std_logic;
           evento: out std_logic_vector(numbitsetiqueta+2-1 downto 0);
           concedidaescritura: in std_logic );
end ur;

architecture urarch of ur is


    component contador
       generic( n : integer := 5);
       port( clk, rst, ld, dec : in std_logic;
             din : in std_logic_vector( n-1 downto 0 );
             dout : out std_logic_vector( n-1 downto 0 ) );
    end component;

    component registro
       generic( n : integer := 8);
       port( clk, rst, ld : in std_logic;
             din : in std_logic_vector( n-1 downto 0 );
             dout : out std_logic_vector( n-1 downto 0 ) );
```

```
    end component;

    component fifo16bits
       generic (numentradaslog : integer := 3 )
       port (entradadatos: in std_logic_vector(15 downto 0);
         clk, rst, leer, escribir, enable : in std_logic;
         salidadatos: out std_logic_vector(15 downto 0);
         fifollena, fifovacia: out std_logic);
    end component;

    type states is (lee_fifo, finalizada, cargando_conf, esperando_ejecucion,
           ejecutando, escribirfinejecucion, escribirfinconfiguracion);

    signal currentstate, nextstate: states;

    signal salidafifo: std_logic_vector(longitud-1 downto 0);
    signal salidareg: std_logic_vector(numbitsetiqueta-1 downto 0);
    signal iguales, avanzar, ld_registro, tabla_vacia: std_logic;
    signal ceros, uno: std_logic_vector(long_eje_carga-1 downto 0);
    signal cargada, ejecutada: std_logic;
    signal dec_carga, dec_eje: std_logic;
    signal salida_cont_carga,
           salida_cont_eje: std_logic_vector(long_eje_carga-1 downto 0);

begin

    ceros <= (others =>'0');
    uno <= ceros + 1;

    fifo1: fifo13bits generic map(numentradaslog)
           port map(entrada_fifo, clk, rst, avanzar, escribir, enable,
                           salidafifo, tabla_llena, tabla_vacia);

    registrofifo: registro generic map(numbitsetiqueta)
           port map(clk, rst, ld_registro, salidafifo(12 downto 10),
                           salidareg);

    cont_eje: contador generic map(long_eje_carga)
           port map(clk, rst, comenzar_ejecucion, dec_eje,
                salidafifo(2*long_eje_carga-1 downto long_eje_carga) ,
                salida_cont_eje);

    cont_carga: contador generic map(long_eje_carga)
           port map(clk, rst, comenzar_carga, dec_carga,
                salidafifo(long_eje_carga-1 downto 0) , salida_cont_carga);


    cargada <= '1' when (salida_cont_carga = uno ) else '0';
    ejecutada <= '1'    when (salida_cont_eje = uno ) else '0';

    dec_carga <= not cargada;
    dec_eje <= not ejecutada;

    iguales <= '1' when (salidareg = salidafifo(12 downto 10)) else '0';
    sub_actual <= salidafifo;
    vacia <= tabla_vacia;

     --control de la tabla

    stategen:
    process (escribir, iguales, tabla_vacia, currentstate, comenzar_carga,
                  cargada, comenzar_ejecucion, ejecutada, concedidaescritura)
    begin
      nextstate <= currentstate;
      case currentstate is

       when lee_fifo =>
                if (tabla_vacia='0' and escribir='0') then
                    nextstate<=finalizada;
                end if;
```

```
            when finalizada =>
                if (iguales='0' and comenzar_carga='1') then
                    nextstate<=cargando_conf;
                elsif (iguales='1') then
                    nextstate<=escribirfinconfiguracion;
                end if;

            when cargando_conf =>
                if (cargada='1') then
                    nextstate<=escribirfinconfiguracion;
                end if;

            when escribirfinconfiguracion =>
                if (concedidaescritura='1') then
                    nextstate<=esperando_ejecucion;
                end if;

            when esperando_ejecucion =>
                if (comenzar_ejecucion='1') then
                    nextstate<=ejecutando;
                end if;

            when ejecutando =>
                if (ejecutada='1') then
                    nextstate<=escribirfinejecucion;
                end if;

            when escribirfinejecucion =>
                if (concedidaescritura='1') then
                    nextstate<=lee_fifo;
                end if;

        end case;
    end process stategen;

    state:
    process (rst, clk)
    begin
      if (rst = '1') then
        currentstate <= lee_fifo;
      elsif (clk'event and clk='1') then
        currentstate <= nextstate;
      end if;
    end process state;

    evento(numbitsetiqueta+2-1 downto 2)<=
                 salidafifo(longitud-1 downto longitud-numbitsetiqueta);

    mooremealygen:
    process (escribir, iguales, tabla_vacia, currentstate, comenzar_carga,
     cargada, comenzar_ejecucion, ejecutada, concedidaescritura, salidafifo)
    begin
      case currentstate is

     when lee_fifo =>
                avanzar <= tabla_vacia nor escribir;
                pedir_carga<='0';
                pedir_ejecucion<='0';
                ld_registro<='0';
                estado<="000";
                escribirenfifoeventos <= '0';
                evento(1 downto 0)<="00";

            when finalizada =>
                pedir_carga <= not iguales;
                pedir_ejecucion<='0';
                avanzar<='0';
                ld_registro<='0';
                estado<="001";
```

```
                escribirenfifoeventos <= '0';
                evento(1 downto 0) <= "00";

            when cargando_conf =>
                pedir_ejecucion<='0';
                pedir_carga<='0';
                avanzar<='0';
                ld_registro<='0';
                estado<="010";
                escribirenfifoeventos <= '0';
                evento(1 downto 0) <= "00";

            when esperando_ejecucion =>
                pedir_ejecucion<='1';
                pedir_carga<='0';
                avanzar<='0';
                ld_registro<='1'; --copio la salida de la fifo en el registro
                estado<="100";
                escribirenfifoeventos <= '0';
                evento(1 downto 0) <= "00";

            when ejecutando =>
                pedir_ejecucion<='0';
                pedir_carga<='0';
                avanzar<='0';
                ld_registro<='0';
                estado<="101";
                escribirenfifoeventos <= '0';
                evento(1 downto 0) <= "00";

            when escribirfinejecucion =>
                pedir_ejecucion<='0';
                pedir_carga<='0';
                avanzar<='0';
                ld_registro<='0';
                estado<="110";
                escribirenfifoeventos <= '1';
                evento(1 downto 0) <= "10";

            when escribirfinconfiguracion =>
                pedir_ejecucion<='0';
                pedir_carga<='0';
                avanzar<='0';
                ld_registro<='0';
                estado<="011";
                escribirenfifoeventos <= '1';
                evento(1 downto 0) <= "01";

        end case;
    end process mooremealygen;

end urarch;
```

IterativaBásica.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity iterativaBasica is
   port( clk,rst,libre_ant, borrar, acierto_i, load_tarea : in std_logic;
         load_tarea_i,libre_sig : out std_logic);
end iterativaBasica;

architecture iterativaBasicaArch of iterativaBasica is

component biestable is
```

```
    port( clk, rst, ld : in std_logic;
       din : in std_logic;
       dout : out std_logic);
end component;

    signal salida_libre,salida_puertaAND3: std_logic;
    signal entradaBiestable, cargaBiestable: std_logic;

begin

  librei: biestable port map
                (clk,rst,cargaBiestable,entradaBiestable,salida_libre);


    entradaBiestable <= borrar and acierto_i;
    cargaBiestable <= salida_puertaAND3 or (borrar and acierto_i);
    salida_puertaAND3 <= (not libre_ant) and salida_libre and load_tarea;
    load_tarea_i <= salida_puertaAND3;

    libre_sig <= libre_ant or salida_libre;

end iterativaBasicaArch;
```

EntradaTablaTareas.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity entradaTablaTareas is
    generic( numBitsContador : integer := 3;
                numBitsEtiqueta : integer := 3);
    port (
        clk, rst, loadTarea, dependenciaResuelta : in std_logic;
        prepararaParaEjecutar, acierto : out std_logic;
        etiquetaIn : in std_logic_vector(numBitsEtiqueta-1 downto 0);
        --LA ENTRADA ES: Etiqueta--numSucesores--numPredecesores--subtareas
        entradaDatos : in std_logic_vector
                    ( numBitsContador+numBitsEtiqueta*(2**numBitsContador)
                                                +numBitsEtiqueta-1 downto 0);
        --LA SALIDA ES: numSucesores--subtareas
        salidaDatos : out std_logic_vector(numBitsEtiqueta
                      +numBitsEtiqueta*(2**numBitsContador-1)-1 downto 0) );
end entradaTablaTareas;

architecture entradaTablaTareasArch of entradaTablaTareas is

component registro
   generic( n : integer := numBitsEtiqueta );
   port( clk, rst, ld : in std_logic;
      din : in std_logic_vector( n-1 downto 0 );
      dout : out std_logic_vector( n-1 downto 0 ) );
end component;

component contador
   generic( n : integer := numBitsContador );
   port( clk, rst, ld, dec : in std_logic;
      din : in std_logic_vector( n-1 downto 0 );
      dout : out std_logic_vector( n-1 downto 0 ) );
end component;


  signal salidaEtiquetaEntrada:
                          std_logic_vector (numBitsEtiqueta-1 downto 0);
  signal salidaContadorPredecesores, cero:
                          std_logic_vector (numBitsContador-1 downto 0);
```

```
    signal decrementar: std_logic;
    signal salidaComparador, hayCeroPredecesores: std_logic;

  begin

    cero <= (others =>'0');

    etiquetaEntrada: registro generic map (numBitsEtiqueta)
                     port map (clk=>clk,
                               rst=>rst,
                               ld=>loadTarea,
                               din=>
                                 entradaDatos(numBitsContador+numBitsEtiqueta*
                                 (2**numBitsContador)+numBitsEtiqueta-1
                                 downto
                                 numBitsContador+numBitsEtiqueta*(2**
                                 numBitsContador)),
                               dout=>salidaEtiquetaEntrada);

    registroNumSucesores: registro generic map (numBitsContador)
                     port map (clk=>clk,
                               rst=>rst,
                               d=>loadTarea,din=>entradaDatos(numBitsContador+
                                   numBitsEtiqueta*(2**numBitsContador)-1
                                   downto
                                   numBitsContador+numBitsEtiqueta*(2**
                                   numBitsContador-1)),
                               dout=>salidaDatos(
                                  numBitsEtiqueta+numBitsEtiqueta*
                                  (2**numBitsContador-1)-1
                                  downto
                                   numBitsEtiqueta*(2**numBitsContador-1)) );

    contadorPredecesores: contador generic map (numBitsContador)
                          port map (clk=>clk,
                                    rst=>rst,
                                    ld=>loadTarea,
                                    dec=>decrementar,
                                    din=>entradaDatos(numBitsContador+
                                        numBitsEtiqueta*(2**numBitsContador
                                        -1)-1
                                        downto
                                        numBitsEtiqueta*(2**numBitsContador-1)),
                                    dout=>salidaContadorPredecesores);


    subtareas : for I in 0 to 2**numBitsContador-2 generate
    subtarea : registro generic map (numBitsEtiqueta)
                  port map (clk=>clk,
                            rst=>rst,
                            ld=>loadTarea,
                            din=>entradaDatos(I*numBitsEtiqueta+
                                  numBitsEtiqueta-1 downto I*numBitsEtiqueta),
                            dout=>salidaDatos(I*numBitsEtiqueta+
                                  numBitsEtiqueta-1 downto I*numBitsEtiqueta));
    end generate;


    salidaComparador <= '1' when (salidaEtiquetaEntrada = etiquetaIn) else '0';
    hayCeroPredecesores <= '1' when (salidaContadorPredecesores = cero)
                                                                  else '0';

    prepararaparaejecutar <=  hayCeroPredecesores and salidaComparador;
    acierto <= salidaComparador;
    decrementar <= salidaComparador and dependenciaResuelta;

  end entradaTablaTareasArch;
```

TablaTareas.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity tablatareas is
    generic( numbitscontador : integer := 3;
             numbitsetiqueta : integer := 3;
             numentradaslog : integer := 3 );
    port (
        clk, rst, loadtarea, actualizardependencias, borrar : in std_logic;
        prepararaparaejecutar, acierto, tablallena : out std_logic;
        etiquetain : in std_logic_vector(numbitsetiqueta-1 downto 0);
        entradadatos : in
                        std_logic_vector(numbitscontador+numbitsetiqueta*(2*
                        *numbitscontador)+numbitsetiqueta-1 downto 0);
        salidadatos : out
                        std_logic_vector(numbitsetiqueta+numbitsetiqueta*(2*
                        *numbitscontador-1)-1 downto 0);
        numentradasocupadas : out std_logic_vector(numentradaslog downto 0));
end tablatareas;

architecture tablatareasarch of tablatareas is


component entradatablatareas
    generic( numbitscontador : integer := 3;
             numbitsetiqueta : integer := 3 );
    port (
        clk, rst, loadtarea, dependenciaresuelta : in std_logic;
        prepararaparaejecutar, acierto : out std_logic;
        etiquetain : in std_logic_vector(numbitsetiqueta-1 downto 0);
        entradadatos : in
                std_logic_vector(numbitscontador+numbitsetiqueta*(2**numbits
                contador)+numbitsetiqueta-1 downto 0);
        salidadatos : out
                std_logic_vector(numbitsetiqueta+numbitsetiqueta*(2**numbits
                contador-1)-1 downto 0) );
end component;

component priorityencoder
    generic( n : integer := numentradaslog );
    port(
       x : in std_logic_vector( 2**n-1 downto 0 );
       y : out std_logic_vector ( n-1 downto 0 );
       gs : out std_logic );
end component;

component iterativabasica
   port( clk,rst,libre_ant, borrar, acierto_i, load_tarea : in std_logic;
       load_tarea_i,libre_sig : out std_logic);
end component;

component contadorincdec
   generic( n : integer := 8 );
   port( clk, rst, inc, dec : in std_logic;
      dout : out std_logic_vector( n-1 downto 0 ) );
end component;

  signal preparadas, aciertos: std_logic_vector
                                        ((2**numentradaslog)-1 downto 0);
  signal libre: std_logic_vector ((2**numentradaslog) downto 0);
  signal loadtareain: std_logic_vector ((2**numentradaslog)-1 downto 0);
  signal salidacodificador: std_logic_vector (numentradaslog-1 downto 0);
  signal salidasdatos: std_logic_vector
```

```
                    ((numbitsetiqueta+numbitsetiqueta*(2**numbitscontador-1))
                    *(2**numentradaslog)-1 downto 0);
   signal salidadatosintermedia:std_logic_vector
                    (numbitsetiqueta+numbitsetiqueta*(2**numbitscontador-1)-1
                     downto 0);
   signal cero: std_logic_vector (numbitscontador-1 downto 0);
   signal algunavaleuno: std_logic;

begin

    cero <= (others=>'0');

    tabla : for i in 0 to (2**numentradaslog)-1 generate
       subtarea : entradatablatareas
            generic map (numbitscontador, numbitsetiqueta)
             port map(clk=>clk,
                      rst=>rst,
                      loadtarea=>loadtareain(i),
                      dependenciaresuelta=>actualizardependencias,
                      prepararaparaejecutar=>preparadas(i),
                      acierto=>aciertos(i),
                      etiquetain=>etiquetain,
                      entradadatos=>entradadatos,
                      salidadatos=>
                          salidasdatos((numbitsetiqueta+numbitsetiqueta*
                          (2**numbitscontador-1))
                          *i+(numbitsetiqueta+numbitsetiqueta*
                          (2**numbitscontador-1))-1
                          downto
                          (numbitsetiqueta+numbitsetiqueta*
                          (2**numbitscontador-1))*i ) );
    end generate;

    libre(0) <= '0';

    rediterativa : for i in 0 to (2**numentradaslog)-1 generate
       celdabasica: iterativabasica
          port map ( clk, rst, libre(i), borrar, aciertos(i), loadtarea,
                     loadtareain(i), libre(i+1) );
    end generate;

    prepararaparaejecutar <= '0' when (preparadas = cero) else '1';

    codificador: priorityencoder generic map (numentradaslog)
          port map (aciertos, salidacodificador, algunavaleuno);

    salidadatosintermedia <= salidasdatos(
                  (numbitsetiqueta*(2**numbitscontador-1)+numbitsetiqueta)
                  *conv_integer(salidacodificador)+numbitsetiqueta*(2**
                  numbitscontador-1)+numbitsetiqueta-1
                  downto
                  (numbitsetiqueta*(2**numbitscontador-1)+numbitsetiqueta)*
                  conv_integer(salidacodificador) );

    salidadatos <= salidadatosintermedia;
    tablallena <= not libre(2**numentradaslog);
    acierto <= '0' when (aciertos = cero) else '1';

    contadorentradasocupadas : contadorincdec generic map (numentradaslog+1)
       port map (clk=>clk,
                 rst=>rst,
                 inc=>loadtarea, --se debe controlar fuera que no este llena
                 dec=>borrar, --se controla fuera que el borrado sea correcto
                 dout=>numentradasocupadas );

end tablatareasarch;
```

FifoControlTablaTareas.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity fifoControlTablaTareas is
    generic( numBitsContador : integer := 3;
                numBitsEtiqueta : integer := 3;
                numEntradasLogTabla : integer := 3;
                numEntradasLogFifo : integer := 3 );
    port (
        clk, rst, borrar : in std_logic;
        prepararaParaEjecutar, acierto,
                 preparadaTablaSiguienteOperacion : out std_logic;
        etiquetaIn : in std_logic_vector(numBitsEtiqueta-1 downto 0);
        entradaDatosFifo : in
                 std_logic_vector(numBitsContador+numBitsEtiqueta*(2**
                 numBitsContador-1)+numBitsEtiqueta+numBitsEtiqueta-1
                 downto 0);
        escribirFifo: in std_logic;
        fifoSubtareasLlena: out std_logic );
end fifoControlTablaTareas;

architecture fifoControlTablaTareasArch of fifoControlTablaTareas is

component tablaTareas
    generic( numBitsContador : integer := 3;
                numBitsEtiqueta : integer := 3;
                numEntradasLog : integer := 3 );
    port (
        clk, rst, loadTarea, actualizarDependencias, borrar : in std_logic;
        prepararaParaEjecutar, acierto, tablaLlena : out std_logic;
        etiquetaIn : in std_logic_vector(numBitsEtiqueta-1 downto 0);
        entradaDatos : in
           std_logic_vector(numBitsContador+numBitsEtiqueta*(2**
                            numBitsContador)+numBitsEtiqueta-1 downto 0);
         salidaDatos : out
           std_logic_vector(numBitsEtiqueta+numBitsEtiqueta*(2**
                            numBitsContador-1)-1 downto 0);
        numEntradasOcupadas : out std_logic_vector(numEntradasLog downto 0));
end component;

component registro
   generic( n : integer := numBitsEtiqueta );
   port( clk, rst, ld : in std_logic;
      din : in std_logic_vector( n-1 downto 0 );
      dout : out std_logic_vector( n-1 downto 0 ) );
end component;

component contador
   generic( n : integer := numBitsContador );
   port( clk, rst, ld, dec : in std_logic;
      din : in std_logic_vector( n-1 downto 0 );
      dout : out std_logic_vector( n-1 downto 0 ) );
end component;

component fifo30Bits
     generic (numEntradasLog : integer := 3 );
     port (entradaDatos: in std_logic_vector(29 downto 0);
                  clk, rst, leer, escribir, enable : in std_logic;

     salidaDatos: out std_logic_vector(29 downto 0);
     fifoLlena, fifoVacia: out std_logic);
end component;

   type states is (reposo, esperando, cargaSubtareaPrimera,
```

```
      cargaSubtareaContinua, cargaSubtareaFinal, lecturaPrimeraPalabraCarga,
      lecturaPrimeraPalabraLectura);
   signal currentState, nextState: states;

   type statesConsultaActualizarDependencias is
                  (reposoConsultaActualizarDependencias, carga, actualizar);
   signal currentStateConsultaActualizarDependencias,
        nextStateConsultaActualizarDependencias:
        statesConsultaActualizarDependencias;

   signal etiqueta : std_logic_vector(numBitsEtiqueta-1 downto 0);
   signal salidaContador, ceroNumBitsContador, unoNumBitsContador:
                        std_logic_vector (numBitsContador-1 downto 0);
   signal ceroNumEntradasLogFifo, unoNumEntradasLogFifo :
                        std_logic_vector (numEntradasLogFifo-1 downto 0);
   signal salidaRegistroAumentada :
                        std_logic_vector (numBitsEtiqueta*(2**
                                            numBitsContador)-1 downto 0);
   signal decrementarContador, dependenciaResuelta, loadContadorRegistro,
                                            borrarControlador: std_logic;
   signal salidaDatosIntermedia: std_logic_vector(numBitsEtiqueta+
                        numBitsEtiqueta*(2**numBitsContador-1)-1 downto 0);
   signal numEntradasOcupadas : std_logic_vector(numEntradasLogTabla
                                                              downto 0);
   signal aciertoIntermedia : std_logic;
   signal fifoLlena, fifoVacia, enableFifo, leerFifoMaquinaEstados,
                               escribirFifoControl, loadTarea : std_logic;
   signal salidaDatosFifo : std_logic_vector(numBitsContador+numBitsEtiqueta
                                *(2**numEntradasLogTabla-1)+
                                numBitsEtiqueta+numBitsEtiqueta-1 downto 0);

   signal tablaLlena: std_logic;

   signal salidaNumeroNodos: std_logic_vector(numEntradasLogFifo-1 downto 0);
   signal loadContadorNumeroNodos, decrementarNumeroNodos, uno : std_logic;
   signal salidaRegistroEtiqueta: std_logic_vector(numBitsEtiqueta-1
                                                              downto 0);

begin

    uno <= '1';

    ceroNumBitsContador <= (others=>'0');
    unoNumBitsContador <= ceroNumBitsContador+1;

    ceroNumEntradasLogFifo <= (others=>'0');
    unoNumEntradasLogFifo <= ceroNumEntradasLogFifo+1;

    fifoSubtareasLlena <= fifoLlena;
    escribirFifoControl <= escribirFifo and not fifoLlena;
    enableFifo <= '1';

    fifoTabla : fifo30Bits generic map(numEntradasLogFifo)
                     port map (entradaDatos=>entradaDatosFifo,
                     clk=>clk,
                     rst=>rst,
                     leer=>leerFifoMaquinaEstados,
                     escribir=>escribirFifoControl,
                     enable=>enableFifo,
                     salidaDatos=>salidaDatosFifo,
                     fifoLlena=>fifoLlena,
                     fifoVacia=>fifoVacia );


    contadorNumeroNodos: contador
            generic map (numEntradasLogFifo)
             port map (clk=>clk,
             rst=>rst,
```

```
            ld=>loadContadorNumeroNodos,
            dec=>decrementarNumeroNodos,
            din=>salidaDatosFifo(numBitsContador+
               numBitsEtiqueta*(2** numEntradasLogTabla-)+numBitsEtiqueta+
               numBitsEtiqueta-1
               downto
               numBitsContador+numBitsEtiqueta* (2**numEntradasLogTabla-1)+
               numBitsEtiqueta+numBitsEtiqueta- numEntradasLogFifo),
            dout=>salidaNumeroNodos);

   tabla : tablaTareas
      generic map (numBitsContador, numBitsEtiqueta, numEntradasLogTabla)
       port map(clk=>clk,
                rst=>rst,
                loadTarea=>loadTarea,
                actualizarDependencias=>dependenciaResuelta,
                borrar=> borrarControlador,
                prepararaParaEjecutar=> prepararaParaEjecutar,
                acierto=>aciertoIntermedia,
                tablaLlena=> tablaLlena,
                etiquetaIn=>etiqueta,
                entradaDatos=>salidaDatosFifo
                salidaDatos=>salidaDatosIntermedia,
                numEntradasOcupadas=>numEntradasOcupadas );
                acierto <= aciertoIntermedia;

   --Control de la tabla
   stateGen:
        process (escribirFifo, currentState, salidaNumeroNodos,
                 numEntradasOcupadas, fifoVacia, tablaLlena)
   begin
     nextState <= currentState;
     case currentState is

    when reposo =>
        if (salidaNumeroNodos = ceroNumEntradasLogFifo and fifoVacia='0')
                        then nextState <= lecturaPrimeraPalabraLectura;
        end if;

    when lecturaPrimeraPalabraLectura =>
        if (escribirFifo='0') then
           nextState <= lecturaPrimeraPalabraCarga;
        end if;

    when lecturaPrimeraPalabraCarga =>
        if (2**numEntradasLogTabla-numEntradasOcupadas >=
            salidaNumeroNodos  and salidaNumeroNodos /=
            ceroNumEntradasLogFifo and fifoVacia='0' and
            tablaLlena='0') then
                  nextState <= cargaSubtareaPrimera;
        else
            nextState <= esperando;
        end if;

          when esperando =>
        if (2**numEntradasLogTabla-numEntradasOcupadas >=
            salidaNumeroNodos and salidaNumeroNodos /=
             ceroNumEntradasLogFifo and fifoVacia='0' and tablaLlena='0')
             then nextState <= cargaSubtareaPrimera;
        end if;

           when cargaSubtareaPrimera =>
         if (escribirFifo='0') then
             if (salidaNumeroNodos = unoNumEntradasLogFifo) then
                  nextState <= cargaSubtareaFinal;
             else
                 nextState <= cargaSubtareaContinua;
             end if;
```

```
        end if;

          when cargaSubtareaContinua =>
            if (escribirFifo='0') then
               if (salidaNumeroNodos = unoNumEntradasLogFifo) then
                   nextState <= cargaSubtareaFinal;
               end if;
            end if;

           when cargaSubtareaFinal => nextState <= reposo;

 end case;
end process stateGen;

state:
process (rst, clk)
begin
  if (rst = '1') then
    currentState <= reposo;
  elsif (clk'event and clk='1') then
    currentState <= nextState;
  end if;
end process state;

mooreGen:
process (escribirFifo, currentState)
begin
  case currentState is

 when reposo =>
    loadTarea <='0';
    decrementarNumeroNodos <= '0';
    leerFifoMaquinaEstados <= '0';
          loadContadorNumeroNodos <= '0';

      when esperando =>
    loadTarea <='0';
    decrementarNumeroNodos <= '0';
    leerFifoMaquinaEstados <= '0';
          loadContadorNumeroNodos <= '0';

 when cargaSubtareaPrimera =>
   loadTarea <='0';
   decrementarNumeroNodos <= not escribirFifo;
   leerFifoMaquinaEstados <= not escribirFifo;
         loadContadorNumeroNodos <= '0';

 when cargaSubtareaContinua =>
   loadTarea <= not escribirFifo;
   decrementarNumeroNodos <= not escribirFifo;
   leerFifoMaquinaEstados <= not escribirFifo;
         loadContadorNumeroNodos <= '0';

 when cargaSubtareaFinal =>
   loadTarea <='1';
   decrementarNumeroNodos <= '0';
   leerFifoMaquinaEstados <= '0';
         loadContadorNumeroNodos <= '0';

 when lecturaPrimeraPalabraLectura =>
   loadTarea <='0';
   decrementarNumeroNodos <= '0';
   leerFifoMaquinaEstados <= not escribirFifo;
         loadContadorNumeroNodos <= '0';

 when lecturaPrimeraPalabraCarga =>
   loadTarea <='0';
   decrementarNumeroNodos <= '0';
```

```
        leerFifoMaquinaEstados <= '0';
              loadContadorNumeroNodos <= '1';

     end case;
    end process mooreGen;


    -- Esto es para actualizar las dependencias
    stateGenConsultaActualizarDependencias:
    process (currentStateConsultaActualizarDependencias, borrar,
                  salidaContador, salidaDatosIntermedia, aciertoIntermedia)
    begin
      nextStateConsultaActualizarDependencias <=
                               currentStateConsultaActualizarDependencias;
      case currentStateConsultaActualizarDependencias is

        when reposoConsultaActualizarDependencias =>
          if (borrar ='1'  and aciertoIntermedia ='1') then
               nextStateConsultaActualizarDependencias <= carga;
          end if;

        when carga =>
          if (salidaDatosIntermedia(numBitsEtiqueta*(2**numBitsContador-
             1)+numBitsEtiqueta-1 downto
             numBitsEtiqueta*(2**numBitsContador-1)+numBitsEtiqueta-
             numBitsContador) = ceroNumBitsContador)
             then
               nextStateConsultaActualizarDependencias <=
               reposoConsultaActualizarDependencias;
             else
               nextStateConsultaActualizarDependencias <= actualizar;
             end if;

       when actualizar =>
         if (salidaContador = unoNumBitsContador) then
            nextStateConsultaActualizarDependencias <=
            reposoConsultaActualizarDependencias;
         end if;

      end case;
    end process stateGenConsultaActualizarDependencias;

     stateConsultaActualizarDependencias: process (rst, clk)
     begin
       if (rst = '1') then
         currentStateConsultaActualizarDependencias <=
                               reposoConsultaActualizarDependencias;
       elsif (clk'event and clk='1') then
         currentStateConsultaActualizarDependencias <=
                               nextStateConsultaActualizarDependencias;
       end if;
     end process stateConsultaActualizarDependencias;

     mooreMealyGenConsultaActualizarDependencias:
     process (currentStateConsultaActualizarDependencias,
             etiquetaIn, salidaRegistroAumentada, salidaContador,
             salidaDatosFifo, borrar, aciertoIntermedia,
             salidaRegistroEtiqueta)
     begin
       case currentStateConsultaActualizarDependencias is

       when reposoConsultaActualizarDependencias =>
             preparadaTablaSiguienteOperacion <= '1';
             borrarControlador <='0';
             dependenciaResuelta <= '0';
             decrementarContador <= '0';
             etiqueta <= salidaRegistroEtiqueta;
```

```
        when carga =>
             preparadaTablaSiguienteOperacion <= '0';
             borrarControlador <='1';
             dependenciaResuelta <= '0';
             loadContadorRegistro <= '1';
             decrementarContador <= '0';
             etiqueta <= salidaRegistroEtiqueta;

        when actualizar =>
             preparadaTablaSiguienteOperacion <= '0';
             borrarControlador <='0';
             dependenciaResuelta <= '1';
             loadContadorRegistro <= '0';
             decrementarContador <= '1';
             etiqueta <= salidaRegistroAumentada( (numBitsEtiqueta*
                    (CONV_INTEGER(salidaContador))+numBitsEtiqueta-1) downto
                    (numBitsEtiqueta*(CONV_INTEGER(salidaContador))));
      end case;
     end process mooreMealyGenConsultaActualizarDependencias;


    registroEtiqueta :  registro generic map (numBitsEtiqueta)
                 port map (clk=>clk,
                           rst=>rst,
                           ld=>uno,
                           din=>etiquetaIn,
                           dout=>salidaRegistroEtiqueta);

      registroSucesores :  registro
                generic map (numBitsEtiqueta*(2**numBitsContador-1))
                port map (clk=>clk,
                          rst=>rst,
                          ld=>loadContadorRegistro,
                          din=>salidaDatosIntermedia(numBitsEtiqueta*
                                (2**numBitsContador-1)-1 downto 0),
                          dout=>salidaRegistroAumentada(numBitsEtiqueta*
                                (2**numBitsContador)-1 downto
                                numBitsEtiqueta));

      salidaRegistroAumentada (numBitsEtiqueta-1 downto 0) <= (others=>'0');

      contadorSucesores: contador
                 generic map (numBitsContador)
                port map (clk=>clk,
                          rst=>rst,
                          ld=>loadContadorRegistro,
                          dec=>decrementarContador,
                          din=>salidaDatosIntermedia(
                               numBitsEtiqueta*(2**numBitsContador-1)+
                               numBitsEtiqueta-1
                               downto
                               numBitsEtiqueta*(2**numBitsContador-1)+
                               numBitsEtiqueta-numBitsContador),
                          dout=>salidaContador);


  end fifoControlTablaTareasArch;
```

TablaURControlFifoEventos.vhd

```
  library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_arith.all;
  use IEEE.std_logic_unsigned.all;

  entity tablaUFControlFifoEventos is
```

```
    generic( numBitsContadorTablaTareas : integer := 3;
             numBitsEtiqueta : integer := 3;
             numEntradasLogTablaTareas : integer := 3;
             numEntradasLogFifoTablaTareas : integer := 3;
             numUFLog : integer := 2;
             numEntradasLogFifoEventos : integer := 3;
             numEntradasLogFifoUFs : integer := 3;
             numEntradasLogFifoReconfiguraciones : integer := 3 );

    port ( clk, rst : in std_logic;
           entradaDatosFifoTablaTareas : in
             std_logic_vector(numBitsContadorTablaTareas+numBitsEtiqueta*
                     (2**numEntradasLogTablaTareas-1)+numBitsEtiqueta+
                      numBitsEtiqueta-1 downto 0);
           escribirFifoTablaTareas: in std_logic;
           fifoSubtareasLlena: out std_logic;
           entradaDatosFifoUFs: in
              std_logic_vector((2**numUFLog)*(2*numBitsEtiqueta+2*5)-1
                                                          downto 0);
           escribirFifoUFs: in std_logic_vector (2**numUFLog-1 downto 0);
           fifosUFLlenas: out std_logic_vector (2**numUFLog-1 downto 0);
           UFenables: in std_logic_vector (2**numUFLog-1 downto 0);
           entradaDatosFifoReconfiguraciones: in
                  std_logic_vector(numBitsEtiqueta-1 downto 0);
           escribirFifoReconfiguraciones: in std_logic;
           fifoReconfiguracionesFifoLlena: out std_logic;
           fifoReconfiguracionesenable: in std_logic;
           circuitoReconfiguracionLibre : in std_logic;
           eventoNuevaTarea: in std_logic;
           concedidaEscrituraNuevaTarea: out std_logic );
end tablaUFControlFifoEventos;

architecture tablaUFControlFifoEventosArch of tablaUFControlFifoEventos is

component fifoControlTablaTareas
    generic( numBitsContador : integer := 3;
                  numBitsEtiqueta : integer := 3;
                  numEntradasLogTabla : integer := 3;
             numEntradasLogFifo : integer := 3 );
    port (clk, rst, borrar : in std_logic;
          prepararaParaEjecutar, acierto,
                  preparadaTablaSiguienteOperacion : out std_logic;
          etiquetaIn : in std_logic_vector(numBitsEtiqueta-1 downto 0);
          entradaDatosFifo : in
                   std_logic_vector(numBitsContador+numBitsEtiqueta*
                                   (2**numEntradasLogTabla-1)+
                                    numBitsEtiqueta+numBitsEtiqueta-1
                                    downto 0);
          escribirFifo: in std_logic;
          fifoSubtareasLlena: out std_logic );
end component;

component UR
    generic (numBitsEtiqueta:natural:=3;
                  longitud:natural:=30;
                  numEntradasLog:natural:=3;
                  long_eje_carga:natural:=5);
    port ( clk: in std_logic;
           rst: in std_logic;
           enable: in std_logic;
           entrada_fifo:in std_logic_vector(longitud-1 downto 0);
           escribir: in std_logic;
           comenzar_carga : in std_logic;
           comenzar_ejecucion : in std_logic;
           pedir_carga : out std_logic;
           pedir_ejecucion : out std_logic;
           sub_actual : out std_logic_vector(longitud-1 downto 0);
           tabla_llena: out std_logic;
```

```
            estado: out std_logic_vector(2 downto 0);
            vacia: out std_logic;
            escribirEnFifoEventos: out std_logic;
            evento: out std_logic_vector(numBitsEtiqueta+2-1 downto 0);
            concedidaEscritura: in std_logic);
end component;

component fifo3Bits
     generic (numEntradasLog : integer := 3 );
     port (entradaDatos: in std_logic_vector(2 downto 0);
                 clk, rst, leer, escribir, enable : in std_logic;
                 salidaDatos: out std_logic_vector(2 downto 0);
                 fifoLlena, fifoVacia: out std_logic);
end component;

component fifo5Bits
     generic (numEntradasLog : integer := 3 );
     port (entradaDatos: in std_logic_vector(4 downto 0);
                 clk, rst, leer, escribir, enable : in std_logic;
                 salidaDatos: out std_logic_vector(4 downto 0);
                 fifoLlena, fifoVacia: out std_logic);
end component;

component priorityEncoder
    generic( n : integer := 3 );
    port(
       x : in std_logic_vector( 2**n-1 downto 0 );
       y : out std_logic_vector ( n-1 downto 0 );
       gs : out std_logic );
end component;

component decoder
    generic( n : integer := 3 );
    port(
       x : in std_logic_vector( n-1 downto 0 );
       en : in std_logic;
       y : out std_logic_vector ( 2**n-1 downto 0 ) );
end component;

component contadorInc
   generic( n : integer := 8 );
   port( clk, rst, inc : in std_logic;
      dout : out std_logic_vector( n-1 downto 0 ) );
end component;

--Señales para la tabla de tareas
  signal tablaTareasBorrar, tablaTareasPrepararaParaEjecutar,
         tablaTareasAcierto,
         tablaTareasPreparadaTablaSiguienteOperacion : std_logic;
  signal tablaTareasEtiquetaIn : std_logic_vector(numBitsEtiqueta-1
                                                          downto 0);

  --Señales para las UR
  signal UFcomenzar_cargas, UFcomenzar_ejecuciones, UFpedir_cargas,
         UFpedir_ejecuciones, UFvacia, UFescribirEnFifoEventos,
         UFconcedidaEscritura : std_logic_vector (2**numUFLog-1 downto 0);
  signal UFeventos: std_logic_vector((numBitsEtiqueta+2)*(2**numUFLog)-1
                                                          downto 0);
  signal UFestados: std_logic_vector(3*(2**numUFLog)-1 downto 0);
  signal UFsub_actuales :
          std_logic_vector((2**numUFLog)*(2*numBitsEtiqueta+2*5)-1 downto 0);

  --Señales para la fifo de eventos
  signal fifoEventosEntradaDatos, fifoEventosSalidaDatos:
         std_logic_vector(2+numBitsEtiqueta-1 downto 0);
  signal fifoEventosenable, fifoEventosLeer, fifoEventosEscribir,
         fifoEventosFifoLlena, fifoEventosFifoVacia : std_logic;
```

```
    --Señales para la fifo de reconfiguraciones
    signal fifoReconfiguracionesSalidaDatos:
                                    std_logic_vector(numBitsEtiqueta-1 downto
                                    0);
    signal fifoReconfiguracionesLeer, fifoReconfiguracionesFifoVacia :
                                    std_logic;

    --Señales para el codificador de la subtarea a cargar
    signal iguales, igualesEvento: std_logic_vector ((2**numUFLog)-1 downto 0);
    signal codificadorSubtareaACargarSalida :
                                    std_logic_vector (numUFLog-1 downto 0);
    signal codificadorSubtareaACargarAlgunaValeUno,
                                    subtareaACargarComenzarCarga : std_logic;

    --Señales para el codificador de comprobar dependencias
    signal codificadorComprobarDependenciasSalida :
                                    std_logic_vector (numUFLog-1 downto 0);
    signal codificadorComprobarDependenciasAlgunaValeUno,
                            comprobarDependenciasComenzarEjecucion : std_logic;

    --Señales para el controlador
    type states is (reposoOLectura, distinguirEvento,
       lecturaFifoReconfiguracionesEventoNuevaTarea,
       distinguirEstadoUnidadEventoNuevaTarea,
       envioComenzarCargaEventoNuevaTarea,
       comprobarDependenciasEventoFinReconfiguracion,
       subtareaPreparadaEventoFinReconfiguracion,
       envioComenzarEjecucionEventoFinReconfiguracion, actualizarDependencias,
       lecturaFifoReconfiguracionesEventoFinEjecucion,
       distinguirEstadoUnidadEventoFinEjecucion,
       envioComenzarCargaEventoFinEjecucion, bucleFor,
       comprobarDependenciasEventoFinEjecucion,
       subtareaPreparadaEventoFinEjecucion,
       envioComenzarEjecucionEventoFinEjecucion);
    signal currentState, nextState: states;

    --Señales para el contador del bucle
    signal contadorBucleIncremento, bucle, rstContadorBucle: std_logic;
    signal contadorBucleSalida: std_logic_vector (numUFLog-1+1 downto 0);

    signal finEjecucion, finReconfiguracion, nuevaTarea:
                                            std_logic_vector (1 downto 0);

    signal contadorBucleSalidaMayorNumUF :
                                    std_logic_vector (numUFLog-1+1 downto 0);
    signal cerosNumBitsEtiqueta :
                            std_logic_vector (numBitsEtiqueta-1 downto 0);

    --Señales para el biestable RS deboLeerFifoReconfiguracion
    signal deboLeerReset, deboLeerSet, deboLeerSalida : std_logic;

    --Para seleccionar a que unidad enviar el comenzar ejecución
    signal decodificadorComprobarDependenciasEntrada:
                            std_logic_vector (numUFLog-1 downto 0);

    signal contadorBucleSalidaMenosUno: std_logic_vector (numUFLog-1 downto 0);

  begin

     tablaTareas : fifoControlTablaTareas
         generic map( numBitsContadorTablaTareas, numBitsEtiqueta,
                      numEntradasLogTablaTareas, numEntradasLogFifoTablaTareas)
         port map (clk=>clk,
                   rst=>rst,
                   borrar=>tablaTareasBorrar,
                   prepararaParaEjecutar=>tablaTareasPrepararaParaEjecutar,
                   acierto=>tablaTareasAcierto,
                   preparadaTablaSiguienteOperacion=>
```

```
                                tablaTareasPreparadaTablaSiguienteOperacion,
                etiquetaIn=>tablaTareasEtiquetaIn,
                entradaDatosFifo=>entradaDatosFifoTablaTareas,
                escribirFifo=>escribirFifoTablaTareas,
                fifoSubtareasLlena=>fifoSubtareasLlena);

    UFs : for I in 0 to (2**numUFLog)-1 generate
          unidadFuncional: UF
           generic map (numBitsEtiqueta, 2*numBitsEtiqueta+2*5,
                                                numEntradasLogFifoUFs, 5)
           port map ( clk=>clk,
                      rst=>rst,
                      enable=>UFenables(I),
                      entrada_fifo=>entradaDatosFifoUFs(I*(2*
                      numBitsEtiqueta+2*5)+(2*numBitsEtiqueta+2*5)-
                      1 downto I*(2*numBitsEtiqueta+2*5)),
                      escribir=>escribirFifoUFs(I),
                      comenzar_carga=>UFcomenzar_cargas(I),
                      comenzar_ejecucion=>UFcomenzar_ejecuciones(I),
                      pedir_carga=>UFpedir_cargas(I),
                      pedir_ejecucion=>UFpedir_ejecuciones(I),
                      sub_actual=> UFsub_actuales(I*(2*numBitsEtiqueta+2*5)
                                                  +2*numBitsEtiqueta+2*5-1
                                                  downto
                                                  I*(2*numBitsEtiqueta+2*5)),
                      tabla_llena=>fifosUFLlenas(I),
                      estado=>UFestados(I*3+3-1 downto I*3),
                      vacia=>UFvacia(I),
                      escribirEnFifoEventos=>UFescribirEnFifoEventos(I),
                      evento=>UFeventos(I*(numBitsEtiqueta+2)+(numBitsEtiqueta
                                        +2)-1 downto I*(numBitsEtiqueta+2)),
                      concedidaEscritura=>UFconcedidaEscritura(I) );

  iguales(I)<= '1' when
                   UFsub_actuales(I*(2*numBitsEtiqueta+2*5)+numBitsEtiqueta+2*
                   5-1 downto
                   I*(2*numBitsEtiqueta+2*5)+2*5) =
                   fifoReconfiguracionesSalidaDatos else '0';

  igualesEvento(I) <= '1' when
                   UFsub_actuales(I*(2*numBitsEtiqueta+2*5)+2*numBitsEtiqueta+
                   2*5-1 downto I*(2*numBitsEtiqueta+2*5)+numBitsEtiqueta+2*5)
                   = fifoEventosSalidaDatos(2+numBitsEtiqueta-1 downto 2)
                   else '0';

  end generate;



  fifoEventosenable <= '1';

  fifoEventos: fifo5Bits generic map (numEntradasLogFifoEventos)
               port map (entradaDatos=>fifoEventosEntradaDatos,
                         clk=>clk,
                         rst=>rst,
                         leer=>fifoEventosLeer,
                         escribir=>fifoEventosEscribir,
                         enable=>fifoEventosenable,
                         salidaDatos=>fifoEventosSalidaDatos,
                         fifoLlena=>fifoEventosFifoLlena,
                         fifoVacia=>fifoEventosFifoVacia);

   fifoReconfiguraciones: fifo3Bits generic map
                                 (numEntradasLogFifoReconfiguraciones)
                port map (entradaDatos=>entradaDatosFifoReconfiguraciones,
                          clk=>clk,
                          rst=>rst,
                          leer=>fifoReconfiguracionesLeer,
```

```
                        escribir=>escribirFifoReconfiguraciones,
                        enable=>fifoReconfiguracionesenable,
                        salidaDatos=>fifoReconfiguracionesSalidaDatos,
                        fifoLlena=>fifoReconfiguracionesFifoLlena,
                        fifoVacia=>fifoReconfiguracionesFifoVacia);

  tablaTareasEtiquetaIn <=
       fifoEventosSalidaDatos(numBitsEtiqueta+2-1 downto 2) when bucle='0'
       else
       UFsub_actuales(conv_integer(contadorBucleSalida(numUFLog-1 downto 0))*
       (2*numBitsEtiqueta+2*5)+(2*numBitsEtiqueta+2*5)-1 downto
       conv_integer (contadorBucleSalida(numUFLog-1 downto 0))
       *(2*numBitsEtiqueta+2*5)+numBitsEtiqueta+2*5 );

  --Controlador y elementos para el mismo

   contadorBucle:  contadorInc generic map (numUFLog+1)
                         port map(clk=>clk,
                                  rst=>rstContadorBucle,
                                  inc=>contadorBucleIncremento,
                                  dout=>contadorBucleSalida );

   decodificadorSubtareaACargar: decoder generic map (numUFLog)
                         port map(x=>codificadorSubtareaACargarSalida,
                                  en=>subtareaACargarComenzarCarga,
                                  y=>UFcomenzar_cargas);

    codificadorSubtareaACargar: priorityEncoder generic map (numUFLog)
                         port map(x=>iguales,
                                  y=>codificadorSubtareaACargarSalida,
                                  gs=>codificadorSubtareaACargarAlgunaValeUno);

   decodificadorComprobarDependencias: decoder generic map (numUFLog)
                         port map(x=>decodificadorComprobarDependenciasEntrada,
                                  en=>comprobarDependenciasComenzarEjecucion,
                                  y=>UFcomenzar_ejecuciones);

   codificadorComprobarDependencias: priorityEncoder generic map (numUFLog)
                         port map(x=>igualesEvento,
                                  y=>codificadorComprobarDependenciasSalida,
                            gs=>codificadorComprobarDependenciasAlgunaValeUno);

   finEjecucion<="10";
   finReconfiguracion<="01";
   nuevaTarea<="11";

   contadorBucleSalidaMayorNumUF
                           <= conv_std_logic_vector(2**numUFLog,numUFLog+1);

  deboLeerFifoReconfiguracion:
     process( clk, rst, deboLeerReset, deboLeerSet )
        begin
           if (rst='1') then
              deboLeerSalida <= '1';
           elsif (clk'event and clk='1') then
              if (deboLeerReset = '1') then
                deboLeerSalida <= '0';
              elsif (deboLeerSet = '1') then
                deboLeerSalida <= '1';
              else
                deboLeerSalida <= deboLeerSalida;
              end if;
           end if;
     end process;

      stateGen:
      process (escribirFifoReconfiguraciones,
```

```
currentState,fifoEventosFifoVacia, fifoEventosSalidaDatos,
circuitoReconfiguracionLibre, fifoReconfiguracionesFifoVacia,
tablaTareasPreparadaTablaSiguienteOperacion,
codificadorSubtareaACargarAlgunaValeUno, UFestados,
tablaTareasPrepararaParaEjecutar, contadorBucleSalida,
codificadorComprobarDependenciasAlgunaValeUno, tablaTareasAcierto)

begin
  nextState <= currentState;
  case currentState is

        when reposoOLectura =>
           if (fifoEventosFifoVacia='0')  then
                nextState <= distinguirEvento;
           end if;

        when distinguirEvento =>
           if (fifoEventosSalidaDatos(1 downto 0)=nuevaTarea and
                circuitoReconfiguracionLibre='1' and
                fifoReconfiguracionesFifoVacia='0') then
                nextState <= lecturaFifoReconfiguracionesEventoNuevaTarea;
            elsif (fifoEventosSalidaDatos(1 downto 0)=finReconfiguracion
                and tablaTareasPreparadaTablaSiguienteOperacion='0') then
                nextState <= distinguirEvento;
            elsif (fifoEventosSalidaDatos(1 downto 0)=finReconfiguracion
                 and tablaTareasPreparadaTablaSiguienteOperacion='1') then
                nextState <= comprobarDependenciasEventoFinReconfiguracion;
            elsif (fifoEventosSalidaDatos(1 downto 0)=finEjecucion and
                tablaTareasPreparadaTablaSiguienteOperacion='0') then
                nextState <= distinguirEvento;
            elsif (fifoEventosSalidaDatos(1 downto 0)=finEjecucion and
                tablaTareasPreparadaTablaSiguienteOperacion='1') then
                nextState <= actualizarDependencias;
            else
                nextState <= reposoOLectura;
            end if;


  --PRINCIPIO EVENTO NUEVA TAREA

  when lecturaFifoReconfiguracionesEventoNuevaTarea =>
          if (escribirFifoReconfiguraciones='0') then
              nextState <= distinguirEstadoUnidadEventoNuevaTarea;
          end if;

  when distinguirEstadoUnidadEventoNuevaTarea =>
          if (codificadorSubtareaACargarAlgunaValeUno='1' and
               UFestados(conv_integer(codificadorSubtareaACargarSalida)*3+
               3-1 downto conv_integer
               (codificadorSubtareaACargarSalida)*3)="001" and
               UFpedir_cargas(conv_integer
               (codificadorSubtareaACargarSalida))='1') then
                  nextState <= envioComenzarCargaEventoNuevaTarea;
          else
                  nextState <= reposoOLectura;
          end if;

  when envioComenzarCargaEventoNuevaTarea =>
           nextState <= reposoOLectura;


  --FIN EVENTO NUEVA TAREA
  --PRINCIPIO EVENTO FIN RECONFIGURACION

  when comprobarDependenciasEventoFinReconfiguracion =>
                  nextState <= subtareaPreparadaEventoFinReconfiguracion;

  when subtareaPreparadaEventoFinReconfiguracion =>
```

```
      if (codificadorComprobarDependenciasAlgunaValeUno='1'
          and tablaTareasAcierto='1'
          and tablaTareasPrepararaParaEjecutar='1') then
            nextState <= envioComenzarEjecucionEventoFinReconfiguracion;
      elsif (circuitoReconfiguracionLibre='1') then
            nextState <= lecturaFifoReconfiguracionesEventoNuevaTarea;
      else
            nextState <= reposoOLectura;
      end if;

   when envioComenzarEjecucionEventoFinReconfiguracion =>
      if (circuitoReconfiguracionLibre='1') then
         nextState <= lecturaFifoReconfiguracionesEventoNuevaTarea;
      else
         nextState <= reposoOLectura;
      end if;

   --FIN EVENTO FIN RECONFIGURACION
   --PRINCIPIO EVENTO FIN EJECUCION

   when actualizarDependencias =>
      if (circuitoReconfiguracionLibre='1') then
         nextState <= lecturaFifoReconfiguracionesEventoFinEjecucion;
      else
         nextState <= bucleFor;
      end if;


   when lecturaFifoReconfiguracionesEventoFinEjecucion =>
     if (escribirFifoReconfiguraciones='0') then
           nextState <= distinguirEstadoUnidadEventoFinEjecucion;
     end if;

   when distinguirEstadoUnidadEventoFinEjecucion =>
     if (codificadorSubtareaACargarAlgunaValeUno='1' and
        UFestados(conv_integer (codificadorSubtareaACargarSalida)*3+3-1
        downto conv_integer (codificadorSubtareaACargarSalida)*3)="001"
        and UFpedir_cargas
        (conv_integer (codificadorSubtareaACargarSalida))='1') then
           nextState <= envioComenzarCargaEventoFinEjecucion;
     else
           nextState <= bucleFor;
     end if;

   when envioComenzarCargaEventoFinEjecucion =>
     nextState <= bucleFor;

   when bucleFor =>
     if (contadorBucleSalida >= contadorBucleSalidaMayorNumUF) then
           nextState <= reposoOLectura;
     elsif (UFpedir_ejecuciones(conv_integer (contadorBucleSalida(
            numUFLog-1 downto 0)))='1')
           nextState <= comprobarDependenciasEventoFinEjecucion;
     end if;

   when comprobarDependenciasEventoFinEjecucion =>
       if (tablaTareasPreparadaTablaSiguienteOperacion='1') then
           nextState <= subtareaPreparadaEventoFinEjecucion;
       end if;

   when subtareaPreparadaEventoFinEjecucion =>
      if (codificadorComprobarDependenciasAlgunaValeUno='1' and
        tablaTareasAcierto='1'
        and tablaTareasPrepararaParaEjecutar='1') then
          nextState <= envioComenzarEjecucionEventoFinEjecucion;

      else
          nextState <= bucleFor;
```

```
        end if;

      when envioComenzarEjecucionEventoFinEjecucion =>
        nextState <= bucleFor;

            --FIN EVENTO FIN EJECUCION

  end case;
  end process stateGen;

  state:
  process (rst, clk)
  begin
    if (rst = '1') then
      currentState <= reposoOLectura;
    elsif (clk'event and clk='1') then
      currentState <= nextState;
    end if;
  end process state;


  mooreGen:
  process (escribirFifoReconfiguraciones, currentState, fifoEventosFifoVacia,
     deboLeerSalida, fifoReconfiguracionesFifoVacia,
     codificadorComprobarDependenciasSalida,
     contadorBucleSalida(numUFLog-1 downto 0))
   begin
      case currentState is

        when reposoOLectura =>
          fifoEventosLeer <= NOT fifoEventosFifoVacia;
          fifoReconfiguracionesLeer <= '0';
          subtareaACargarComenzarCarga <= '0';
          comprobarDependenciasComenzarEjecucion <= '0';
          tablaTareasBorrar <= '0';
          rstContadorBucle <= '0';
          contadorBucleIncremento <= '0';
          bucle <= '0';
          deboLeerReset <= '0';
          deboLeerSet <= '0';
          decodificadorComprobarDependenciasEntrada <=
                                    codificadorComprobarDependenciasSalida;

      when distinguirEvento =>
          fifoEventosLeer <= '0';
          fifoReconfiguracionesLeer <= '0';
          subtareaACargarComenzarCarga <= '0';
          comprobarDependenciasComenzarEjecucion <= '0';
          tablaTareasBorrar <= '0';
          rstContadorBucle <= '0';
          contadorBucleIncremento <= '0';
          bucle <= '0';
          deboLeerReset <= '0';
          deboLeerSet <= '0';
          decodificadorComprobarDependenciasEntrada <=
                                     codificadorComprobarDependenciasSalida;

      when lecturaFifoReconfiguracionesEventoNuevaTarea =>
         fifoEventosLeer <= '0';
         fifoReconfiguracionesLeer <= deboLeerSalida AND NOT
                                           escribirFifoReconfiguraciones;
         subtareaACargarComenzarCarga <= '0';
         comprobarDependenciasComenzarEjecucion <= '0';
         tablaTareasBorrar <= '0';
         rstContadorBucle <= '0';
         contadorBucleIncremento <= '0';
         bucle <= '0';
         deboLeerReset <= '0';
```

```
            deboLeerSet <= '0';
            decodificadorComprobarDependenciasEntrada <=
                                    codificadorComprobarDependenciasSalida;

        when distinguirEstadoUnidadEventoNuevaTarea =>
            fifoEventosLeer <= '0';
            fifoReconfiguracionesLeer <= '0';
            subtareaACargarComenzarCarga <= '0';
            comprobarDependenciasComenzarEjecucion <= '0';
            tablaTareasBorrar <= '0';
            rstContadorBucle <= '0';
            contadorBucleIncremento <= '0';
            bucle <= '0';
            deboLeerReset <= '1';
            deboLeerSet <= '0';
            decodificadorComprobarDependenciasEntrada <=
                                    codificadorComprobarDependenciasSalida;

        when envioComenzarCargaEventoNuevaTarea =>
            fifoEventosLeer <= '0';
            fifoReconfiguracionesLeer <= '0';
            subtareaACargarComenzarCarga <= '1';
            comprobarDependenciasComenzarEjecucion <= '0';
            tablaTareasBorrar <= '0';
            rstContadorBucle <= '0';
            contadorBucleIncremento <= '0';
            bucle <= '0';
            deboLeerReset <= '0';
            deboLeerSet <= '1';
            decodificadorComprobarDependenciasEntrada <=
                                    codificadorComprobarDependenciasSalida;

        when comprobarDependenciasEventoFinReconfiguracion =>
            fifoEventosLeer <= '0';
            fifoReconfiguracionesLeer <= '0';
            subtareaACargarComenzarCarga <= '0';
            comprobarDependenciasComenzarEjecucion <= '0';
            tablaTareasBorrar <= '0';
            rstContadorBucle <= '0';
            contadorBucleIncremento <= '0';
            bucle <= '0';
            deboLeerReset <= '0';
            deboLeerSet <= '0';
            decodificadorComprobarDependenciasEntrada <=
                                    codificadorComprobarDependenciasSalida;

        when subtareaPreparadaEventoFinReconfiguracion =>
            fifoEventosLeer <= '0';
            fifoReconfiguracionesLeer <= '0';
            subtareaACargarComenzarCarga <= '0';
            comprobarDependenciasComenzarEjecucion <= '0';
            tablaTareasBorrar <= '0';
            rstContadorBucle <= '0';
            contadorBucleIncremento <= '0';
            bucle <= '0';
            deboLeerReset <= '0';
            deboLeerSet <= '0';
            decodificadorComprobarDependenciasEntrada <=
                                    codificadorComprobarDependenciasSalida;

        when envioComenzarEjecucionEventoFinReconfiguracion =>
            fifoEventosLeer <= '0';
            fifoReconfiguracionesLeer <= '0';
            subtareaACargarComenzarCarga <= '0';
            comprobarDependenciasComenzarEjecucion <= '1';
            tablaTareasBorrar <= '0';
            rstContadorBucle <= '0';
            contadorBucleIncremento <= '0';
```

```
            bucle <= '0';
            deboLeerReset <= '0';
            deboLeerSet <= '0';
            decodificadorComprobarDependenciasEntrada <=
                                    codificadorComprobarDependenciasSalida;

         when actualizarDependencias =>
            fifoEventosLeer <= '0';
            fifoReconfiguracionesLeer <= '0';
            subtareaACargarComenzarCarga <= '0';
            comprobarDependenciasComenzarEjecucion <= '0';
            tablaTareasBorrar <= '1';
            rstContadorBucle <= '1';
            contadorBucleIncremento <= '0';
            bucle <= '0';
            deboLeerReset <= '0';
            deboLeerSet <= '0';
            decodificadorComprobarDependenciasEntrada <=
                                    codificadorComprobarDependenciasSalida;

         when lecturaFifoReconfiguracionesEventoFinEjecucion =>
            fifoEventosLeer <= '0';
            fifoReconfiguracionesLeer <= deboLeerSalida and NOT
                                              escribirFifoReconfiguraciones;
            subtareaACargarComenzarCarga <= '0';
            comprobarDependenciasComenzarEjecucion <= '0';
            tablaTareasBorrar <= '0';
            rstContadorBucle <= '0';
            contadorBucleIncremento <= '0';
            bucle <= '0';
            deboLeerReset <= '0';
            deboLeerSet <= '0';
            decodificadorComprobarDependenciasEntrada <=
                                         codificadorComprobarDependenciasSalida;

         when distinguirEstadoUnidadEventoFinEjecucion =>
            fifoEventosLeer <= '0';
            fifoReconfiguracionesLeer <= '0';
            subtareaACargarComenzarCarga <= '0';
            comprobarDependenciasComenzarEjecucion <= '0';
            tablaTareasBorrar <= '0';
            rstContadorBucle <= '0';
            contadorBucleIncremento <= '0';
            bucle <= '0';
            deboLeerReset <= '1';
            deboLeerSet <= '0';
            decodificadorComprobarDependenciasEntrada <=
                                         codificadorComprobarDependenciasSalida;

         when envioComenzarCargaEventoFinEjecucion =>
            fifoEventosLeer <= '0';
            fifoReconfiguracionesLeer <= '0';
            subtareaACargarComenzarCarga <= '1';
            comprobarDependenciasComenzarEjecucion <= '0';
            tablaTareasBorrar <= '0';
            rstContadorBucle <= '0';
            contadorBucleIncremento <= '0';
            bucle <= '0';
            deboLeerReset <= '0';
            deboLeerSet <= '1';
            decodificadorComprobarDependenciasEntrada <=
                                         codificadorComprobarDependenciasSalida;

         when bucleFor =>
            fifoEventosLeer <= '0';
            fifoReconfiguracionesLeer <= '0';
            subtareaACargarComenzarCarga <= '0';
            comprobarDependenciasComenzarEjecucion <= '0';
```

```
         tablaTareasBorrar <= '0';
         rstContadorBucle <= '0';
         contadorBucleIncremento <= NOT UFpedir_ejecuciones(CONV_INTEGER(
                                    contadorBucleSalida(numUFLog-1 DOWNTO 0)));
         bucle <= '1';
         deboLeerReset <= '0';
         deboLeerSet <= '0';
         decodificadorComprobarDependenciasEntrada <=
                                 contadorBucleSalida(numUFLog-1 downto 0);

      when comprobarDependenciasEventoFinEjecucion =>
         fifoEventosLeer <= '0';
         fifoReconfiguracionesLeer <= '0';
         subtareaACargarComenzarCarga <= '0';
         comprobarDependenciasComenzarEjecucion <= '0';
         tablaTareasBorrar <= '0';
         rstContadorBucle <= '0';
         contadorBucleIncremento <= '0';
         bucle <= '1';
         deboLeerReset <= '0';
         deboLeerSet <= '0';
         decodificadorComprobarDependenciasEntrada <=
                                 contadorBucleSalida(numUFLog-1 downto 0);

      when subtareaPreparadaEventoFinEjecucion =>
         fifoEventosLeer <= '0';
         fifoReconfiguracionesLeer <= '0';
         subtareaACargarComenzarCarga <= '0';
         comprobarDependenciasComenzarEjecucion <= '0';
         tablaTareasBorrar <= '0';
         rstContadorBucle <= '0';
         contadorBucleIncremento <= '1';
         bucle <= '1';
         deboLeerReset <= '0';
         deboLeerSet <= '0';
         decodificadorComprobarDependenciasEntrada <=
                                 contadorBucleSalida(numUFLog-1 downto 0);

      when envioComenzarEjecucionEventoFinEjecucion =>
         fifoEventosLeer <= '0';
         fifoReconfiguracionesLeer <= '0';
         subtareaACargarComenzarCarga <= '0';
         comprobarDependenciasComenzarEjecucion <= '1';
         tablaTareasBorrar <= '0';
         rstContadorBucle <= '0';
         contadorBucleIncremento <= '0';
         bucle <= '1';
         deboLeerReset <= '0';
         deboLeerSet <= '0';
         decodificadorComprobarDependenciasEntrada <=
                                              contadorBucleSalidaMenosUno;

     end case;
   end process mooreGen;

   contadorBucleSalidaMenosUno <= contadorBucleSalida(numUFLog-1 downto 0)-1;

     cerosNumBitsEtiqueta <= (others => '0');

 --a la fifo de eventos prioridad en la lectura al resto en la escritura
    arbitro:
    process (fifoEventosLeer, fifoEventosFifoLlena, eventoNuevaTarea,
             UFeventos, UFescribirEnFifoEventos)
      begin
        fifoEventosEscribir <= '0';
        fifoEventosEntradaDatos <= (others => '0');
        UFconcedidaEscritura <= (others => '0');
        concedidaEscrituraNuevaTarea <= '0';
```

```
        if (fifoEventosLeer='0') then
         if (fifoEventosFifoLlena='1') then
             fifoEventosEscribir <= '0';
             fifoEventosEntradaDatos <= (others => '0');
             UFconcedidaEscritura <= (others => '0');
              concedidaEscrituraNuevaTarea <= '0';
         else
            if (eventoNuevaTarea='1') then
                fifoEventosEscribir <= '1';
                fifoEventosEntradaDatos <= cerosNumBitsEtiqueta & nuevaTarea;
                UFconcedidaEscritura <= (others => '0');
                 concedidaEscrituraNuevaTarea <= '1';
            else
             --da prioridad a la mas significativa (codificador prioridad)
             for I in UFescribirEnFifoEventos'reverse_range loop
                if UFescribirEnFifoEventos(I)='1' then
                    fifoEventosEscribir <= '1';
             fifoEventosEntradaDatos <=
                     UFeventos(I*(numBitsEtiqueta+2)+(numBitsEtiqueta+2)-1
                                          downto I*(numBitsEtiqueta+2));
             UFconcedidaEscritura <= conv_std_logic_vector(2**I, 2**numUFLog);
             concedidaEscrituraNuevaTarea <= '0';
                 end if;
             end loop;
             end if;
            end if;
         end if;

      end process arbitro;

end tablaUFControlFifoEventosArch;
```

FUENTES EN C DE LA VERSIÓN SW

Tabla software y definición de tipos

```
#include <stdio.h>
#include <stdlib.h>
#include "xparameters.h"
#include "stdio.h"
#include "xutil.h"
#include "xtmrctr.h"
#include "xbasic_types.h"

XTmrCtr counter;
XStatus Status;
Xuint32 t1, t2;

#define NUMNODOS 3
#define NUMMAXSUCESORES 2
#define NUMURS 2


int circuitoDeReconfiguracionLibre = 1;       // Nos dice si el circuito de
reconfiguración está libre(1) u ocupado(0)

struct nodo
{
      unsigned int id;                        // Identificador del nodo
      unsigned int numPrecesodesPorAcabar;    // Número de precesores que
faltan por finalizar
      unsigned int numSucesores;              // Número de sucesores
      unsigned int sucesores[NUMMAXSUCESORES];// Lista de sucesores

};

struct nodo grafo[NUMNODOS];


// DEFINICIÓN DE ESTADOS PARA LAS URs
// (0) --- comenzarReconfiguración ---> (1) --- finReconfiguración ---> (2) --
- comenzarEjecución ---> (3) --- finEjecución ---> (0)


struct unidadReconfigurable
{
      unsigned int estadoUR;
      unsigned int numNodosAsignados;
      unsigned int indiceSubtarea;
      unsigned int ordenSubtareas[NUMNODOS];
};

struct unidadReconfigurable URs[NUMURS];

struct reconf
{
      unsigned int indice;
      unsigned int fifo[NUMNODOS];
};

struct reconf reconfiguraciones;
```

Distinción de eventos

```
void distingueEvento(unsigned int evento, unsigned int subtarea)
{
	// Después de recibir una interrupción venimos aquí

	if (evento == 0)
	{
		eventoNuevaTarea();
	}
	else
	{
		if (evento == 1)
		{
			eventoFinReconfiguracion(subtarea);
		}
		else
		{
			eventoFinEjecucion(subtarea);
		}
	}
}
```

Tratamiento evento nueva tarea

```
void eventoNuevaTarea()
{
	if (circuitoDeReconfiguracionLibre == 1){
		elegirSubtareaACargar();
	}
}

void elegirSubtareaACargar()
{
	 int i;
	 if (reconfiguraciones.indice < NUMNODOS)
	 {
		unsigned int subtareaFifo =
				reconfiguraciones.fifo[reconfiguraciones.indice];

		//cout << "La subtarea siguente a cargar es la " << subtareaFifo
<< "\n";
		for (i=0;i<NUMURS;i++)
		 {
			if (URs[i].indiceSubtarea < URs[i].numNodosAsignados &&
		    subtareaFifo == URs[i].ordenSubtareas[URs[i].indiceSubtarea])
			    {
				//cout << "\t\tEl estado de la UR "<< i <<" es " <<
URs[i].estadoUR <<"\n";
				if(URs[i].estadoUR == 0)
				{
					//cout << "Enviando señal comienzo
reconfiguración subtarea " << subtareaFifo <<"\n";
					xil_printf("Enviando señal comienzo
reconfiguración subtarea %d\n", subtareaFifo);

					URs[i].estadoUR = 1; // Actualizar el estado
de la unidad
					reconfiguraciones.indice++; // Actualizar el
índice de reconfiguraciones
				 }
				break;
			}
		}
	 }
```

```
}
```

Tratamiento evento fin de reconfiguración

```
void eventoFinReconfiguracion(unsigned int subtareaReconfigurada)
{

	// Buscamos la unidad en la que esté
	int k,m;
	for (m=0; m<NUMURS ;m++)
	{
		if (URs[m].indiceSubtarea < URs[m].numNodosAsignados &&
	  URs[m].ordenSubtareas[URs[m].indiceSubtarea] == subtareaReconfigurada)
		{
			k = m;
			break;
		}
	}
	URs[k].estadoUR = 2; // Actualizar el estado de la unidad

	// Si la subtarea recien reconfigurada puede comenzar su ejecución que
lo haga
	if (grafo[subtareaReconfigurada-1].numPrecesodesPorAcabar == 0)
	{
		//cout << "Enviando señal comienzo ejecución subtarea " <<
subtareaReconfigurada <<"\n";
		xil_printf("Enviando señal comienzo ejecución subtarea %d\n",
subtareaReconfigurada);
		URs[k].estadoUR = 3; // Actualizar el estado de la unidad
	}

	if (circuitoDeReconfiguracionLibre == 1)
	{
		elegirSubtareaACargar();
	}
}
```

Tratamiento evento fin de ejecución

```
void eventoFinEjecucion(unsigned int subtareaEjecutada)
{
	// Buscamos la unidad en la que esté
	int k,m,j,i;
	for (m=0; m<NUMURS ;m++)
	{
		if (URs[m].indiceSubtarea < URs[m].numNodosAsignados &&
	      URs[m].ordenSubtareas[URs[m].indiceSubtarea] == subtareaEjecutada)
		{
			k = m;
			break;
		}
	}

	URs[k].estadoUR = 0;	// Actualizar el estado de la unidad
	URs[k].indiceSubtarea++;  // Actualizar subtarea a tratar en esa UR

	// Actualizar dependencias
	for (j=0; j<grafo[subtareaEjecutada-1].numSucesores ;j++)
	{
		grafo[ grafo[subtareaEjecutada-1].sucesores[j]-1
].numPrecesodesPorAcabar--; // Decrementamos el numero de predecesores por
acabar de cada sucesor
	}
```

```
	for (i=0; i<NUMURS ;i++)
	{
		if (URs[i].estadoUR == 2 &&
		    URs[i].indiceSubtarea < URs[i].numNodosAsignados &&
grafo[URs[i].ordenSubtareas[URs[i].indiceSubtarea]].numPrecesodesPorAcabar ==
0 )
		{
			//cout << "Enviando señal comienzo ejecución subtarea " <<
URs[i].ordenSubtareas[URs[i].indiceSubtarea] <<"\n";
			xil_printf("Enviando señal comienzo ejecución subtarea
%d\n",  URs[i].ordenSubtareas[URs[i].indiceSubtarea]);
			URs[i].estadoUR = 1; // Actualizar el estado de la unidad
		}
	}

	if (circuitoDeReconfiguracionLibre == 1)
	{
		elegirSubtareaACargar();
	}
}
```

Ejemplo de creación de un grafo secuencial de 4 nodos y su planificación asociada para dos URs

```
void creaGrafoPruebas()
{	// 1 --> 2 --> 3 --> 4

	int j;
	grafo[0].numPrecesodesPorAcabar = 0;	// Número de precesores que
faltan por finalizar
	grafo[0].numSucesores = 1;		// Número de sucesores
	for (j=0; j<NUMMAXSUCESORES ;j++)
	{
		grafo[0].sucesores[j] = 0;
	}
	grafo[0].sucesores[0] = 2;


	grafo[1].numPrecesodesPorAcabar = 1;	// Número de precesores que
faltan por finalizar
	grafo[1].numSucesores = 1;		// Número de sucesores
	for (j=0; j<NUMMAXSUCESORES ;j++)
	{
		grafo[1].sucesores[j] = 0;
	}
	grafo[1].sucesores[0] = 3;

	grafo[2].numPrecesodesPorAcabar = 1;	// Número de precesores que
faltan por finalizar
	grafo[2].numSucesores = 1;		// Número de sucesores
	for (j=0; j<NUMMAXSUCESORES ;j++)
	{
		grafo[2].sucesores[j] = 0;
	}
	grafo[2].sucesores[0] = 4;

	grafo[3].numPrecesodesPorAcabar = 1;	// Número de precesores que
faltan por finalizar
	grafo[3].numSucesores = 0;		// Número de sucesores
	for (j=0; j<NUMMAXSUCESORES ;j++)
	{
		grafo[3].sucesores[j] = 0;
	}
```

```
        reconfiguraciones.indice = 0;
        reconfiguraciones.fifo[0] = 1;
        reconfiguraciones.fifo[1] = 2;
        reconfiguraciones.fifo[2] = 3;
        reconfiguraciones.fifo[3] = 4;

        URs[0].estadoUR = 0;
        URs[0].indiceSubtarea = 0;
        URs[0].numNodosAsignados = 2;
        URs[0].ordenSubtareas[0] = 1;
        URs[0].ordenSubtareas[1] = 3;

        URs[1].estadoUR = 0;
        URs[1].indiceSubtarea = 0;
        URs[1].numNodosAsignados = 2;
        URs[1].ordenSubtareas[0] = 2;
        URs[1].ordenSubtareas[1] = 4;
}
```