TESIS DOCTORAL

APORTACIONES AL DISEÑO Y A LAS HERRAMIENTAS DE SÍNTESIS AUTOMÁTICA DE DISPOSITIVOS LÓGICOS PROGRAMABLES

Autor: Vicente Baena Lecuyer

Director: Miguel A. Aguirre Echánove

TESIS DOCTORAL

APORTACIONES AL DISEÑO Y A LAS HERRAMIENTAS DE SÍNTESIS AUTOMÁTICA DE DISPOSITIVOS LÓGICOS PROGRAMABLES

por

Vicente Baena Lecuyer

Ingeniero de Telecomunicación por la E.S. de Ingenieros

de la Universidad de Sevilla

Presentada en la

Escuela Superior de Ingenieros

de la

Universidad de Sevilla

para la obtención del

Grado de Doctor Ingeniero de Telecomunicación

Sevilla, Diciembre de 2000

TESIS DOCTORAL

APORTACIONES AL DISEÑO Y A LAS HERRAMIENTAS DE SÍNTESIS AUTOMÁTICA DE DISPOSITIVOS LÓGICOS PROGRAMABLES

Vicente Baena Lecuyer

Sevilla, Diciembre de 2000

"Si me ofreciesen la sabiduría con la condición de guardarla para mí sin comunicarla a nadie; no la querría."
Lucio Anneo Séneca

Agradecimientos

En primer lugar, quiero dar mi más sincero agradecimiento a todo el *equipo FIPSOC*, en particular a José María Insenser, Julio Faura, Ignacio Lacadena, Joan Cabestany, Juan Manuel Moreno y Jordi Madrenas, sin los cuales esta tesis no existiría.

También quiero agradecer la inestimable ayuda aportada por Antonio Torralba, Miguel Ángel Aguirre, Leopoldo García Franquelo, a Jorge Chávez por sus conocimientos de LaTeX, y en general a todos mis compañeros del grupo de Tecnología Electrónica del Departamento de Ingeniería Electrónica de la Universidad de Sevilla.

Gracias a mi mujer, Rosa, por soportarme durante estos años; y a mis padres, por la educación que me han dado.

Resumen de la Tesis

El presente trabajo es consecuencia directa de la participación del Grupo de Tecnología Electrónica de la Universidad de Sevilla en el proyecto ESPRIT 21625 *FIPSOC*. A grandes rasgos, este proyecto consiste en la integración en un solo circuito integrado de un microprocesador, un bloque analógico y una parte digital programable, lo que se conoce como "un sistema en un chip", y contempla las múltiples ventajas de la solución integrada frente a la discreta. El Grupo de Tecnología Electrónica fue encargado del desarrollo de las herramientas de diseño de la parte digital programable, que a partir de la descripción en lenguaje de alto nivel del circuito a realizar, debía configurar los distintos elementos programables internos al dispositivo.

A lo largo de este trabajo, y dadas las especiales características de los dispositivos, se plantearon una serie de problemas que no pudieron ser resueltos mediante el uso de los algoritmos previamente descritos en la literatura tradicional dando lugar a la necesidad de desarrollar nuevos algoritmos para las distintas tareas comunes en las herramientas de diseño de dispositivos lógicos programables como son el empaquetado, colocación y el rutado. Paralelamente al desarrollo de estas herramientas se realizaron estudios sobre la arquitectura tratada dando lugar a diversos resultados, muy prometedores para las FPGAs comerciales actuales.

Debido a la amplia variedad de temas tratados, se ha considerado conveniente organizar esta tesis en capítulos relativos a la distintas etapas realizadas en el diseño de dispositivos programables. En el primer capítulo, se realizará una introducción sobre dispositivos lógicos programables y herramientas de diseño. El capítulo 2 trata la fase de empaquetado donde la lógica del circuito diseñado debe repartirse en los distintos bloques lógicos disponibles del dispositivo. Los estudios realizados sobre la arquitectura interna de estos bloques, muestran la posibilidad de incrementar la rutabilidad de los circuitos en un 13% para arquitecturas similares a FIPSOC, sin aumentar el área de silicio ocupada.

El capítulo 3 aborda el problema de colocación de los bloques lógicos en el dispositivo lógico programable y de las especiales situaciones que se plantean en este tipo de dispositivos SOCs. Se presenta un algoritmo original de colocación, con el que se consiguen resultados un 17% mejores que los publicados hasta la fecha para arquitecturas de FPGAs comerciales.

En el capítulo 4 se presenta la última etapa, el rutado, que consiste en determinar para cada conexión el estado de los distintos elementos programables para realizar las conexiones necesarias entre los bloques lógicos. En él se presenta un estudio sobre la codificación de los bits de configuración de los bloques lógicos para FPGAs multicontexto, consiguiéndose un ahorro en área de silicio de hasta un 17%, sin reducción de la rutabilidad y sin apenas deteriorar la velocidad de funcionamiento del dispositivo. También se presentan un modelo de área especialmente adapatado para dispositivos programables multicontexto, y un eficiente

algoritmo de rutado aplicable a las arquitecturas bajo estudio, el único existente hasta la fecha.

Por último, en el capítulo 5, se presentan los aspectos más relevantes del producto comercial resultado de esta tesis: la herramienta de síntesis automática *FLIPER* incluida en el entorno de desarrollo *FIPSOC CAE TOOLS*. Software para el desarrollo de aplicaciones en la familia FIP-SOC.

Índice General

Capítulo 1

Introducción

En la actualidad, los dispositivos lógicos programables ofrecen una solución de bajo coste para circuitos digitales de pequeña tirada o el desarrollo de prototipos, frente a la fuerte inversión necesaria en la creación de circuitos integrados personalizados. Funcionalmente hablando, las prestaciones alcanzadas con estos dispositivos cubren un amplio espectro de posibilidades (frecuencias de hasta 100MHz, arquitecturas específicas,...).

Debido a la amplia demanda de este tipo de circuitos, en los últimos años ha aparecido una gran cantidad de dispositivos programables de altas prestaciones (Virtex [XIL99], ProAsic [ACT99b], Altera 2K [ALT99],...), por otro lado también han surgido los llamados *"sistemas en un chip"* (SOC), como FIPSOC [SID99a] o FPSLIC [ATM99] en los que se ofrecen prestaciones medias y la gran ventaja radica en la integración en el mismo chip de elementos que generalmente acompañan a la propia FPGA, tales como microprocesadores, convertidores,..., sistemas muy utilizados en aplicaciones de bajas prestaciones, donde se persigue una reducción de costes.

El Grupo de Tecnología Electrónica de la Universidad de Sevilla, formó parte del consorcio que desarrolló el proyecto FIPSOC. En particular, desarrolló las herramientas CAD utilizadas en la programación de la FPGA contenida en FIPSOC. La integración en un solo chip de un microprocesador, un bloque analógico programable y una FPGA, así como la interrelación entre estas tres partes, planteó nuevas situaciones nunca antes tratadas en la literatura. El presente trabajo nace del estudio de dichas situaciones, y recoge algunos resultados obtenidos aplicables a FPGAs con arquitecturas más generales.

Este capítulo se presenta en dos partes bien diferenciadas. En la primera se realiza una introducción a la arquitectura de los dispositivos lógicos programables, definiendo algunos conceptos y una nomenclatura necesaria para los capítulos posteriores. En la segunda se revisan los distintos algoritmos utilizados en el proceso de implementación automática de los dispositivos lógicos programables, analizando las ventajas e inconvenientes de cada uno de ellos.

1.1 Arquitectura de los dispositivos lógicos programables

Existe una gran variedad de dispositivos lógicos programables (FPGA, CPLD, PLD, PAL...), aunque como ya predijera Brown [BRO92a], en los últimos años las FPGAs han recibido un mayor empuje debido a su versatilidad y la baja inversión necesaria en el desarrollo de prototipos. Los resultados que se presentan en esta tesis, sólo son aplicables a las FPGAs (Field Programmable Gate Arrays) que, como veremos a continuación, poseen una arquitectura muy concreta, aunque también pueden ser aplicadas a otros dispositivos programables con arquitectu-

ras similares.

Todas las FPGAs están compuestos por tres componentes fundamentales: bloques lógicos, bloques de entrada salida, y elementos de rutado programables. La estructura más general de estos dispositivos puede verse en la figura 1.1 [ROS93]. Los circuitos se implementan en una FPGA programando cada bloque lógico de tal forma que cada uno de ellos realiza una parte de la funcionalidad total de la lógica, los bloques de entrada y salida se configuran para permitir la comunicación del circuito con el exterior, y los recursos de rutado se programan para realizar las distintas conexiones entre los bloques lógicos y los bloques de entrada y salida.

La configuración de estos dispositivos se realiza en la mayoría de los casos, a través de celdas de memoria estática (SRAM) que controlan elementos como transistores de paso, multiplexores, decodificadores y búfferes triestado, como ocurre en muchas de las FPGAs de Xilinx [XIL99], y las versiones más avanzadas de Actel [ACT99a] y Altera [ALT99]. En otros casos también se usan antifusibles como en algunas de las FPGAs de Actel y dispositivos de puertas flotantes (memorias EPROM, EEPROM, flash) como la nueva familia ProASIC de Actel [ACT99b].

El uso de SRAM es muy común, tanto en FPGAs como en los SOCs, por lo que esta tesis sólo se centrará en el estudio de dispositivos basados en este tipo de tecnología.

A continuación estudiaremos en detalle cada uno de los tres componentes fundamentales.

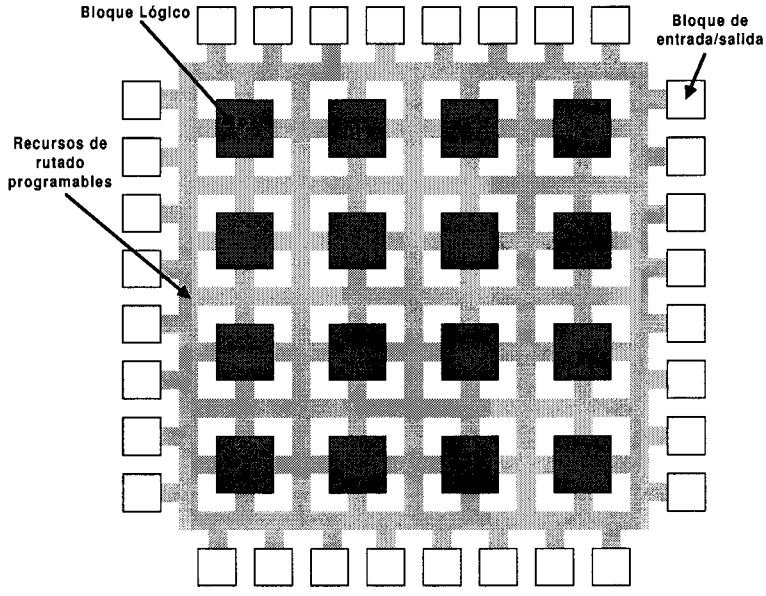


Figura 1.1: Arquitectura general de una FPGA

1.1.1 Bloques lógicos

La arquitectura de los bloques lógicos usados en las FPGAs, influye notablemente en la velocidad y la eficiencia en área del dispositivo, refiriéndose en el caso de la eficiencia en área, al área necesitada por un determinado circuito para ser implementado en una FPGA. Actualmente, las FPGAs tipo SRAM comerciales usan bloques lógicos basados en tablas de verdad, (LUTs o Look-Up Tables) y biestables. Estas LUTs consisten simplemente en la implementación física de una tabla de verdad mediante celdas de memoria y un multiplexor. La figura 1.2 presenta una LUT de dos entradas. Una LUT de k entradas (k-LUT) requiere 2^k celdas de memoria, y un multiplexor de 2^k entradas. Una k-LUT puede implementar cualquier función lógica de k entradas, con sólo programar las 2^k celdas de memoria con la tabla de verdad de la función.

Estudios previos han demostrado que bloques lógicos compuestos por LUTs de 4 entradas, consiguen la mayor eficiencia en área [ROS90]. La

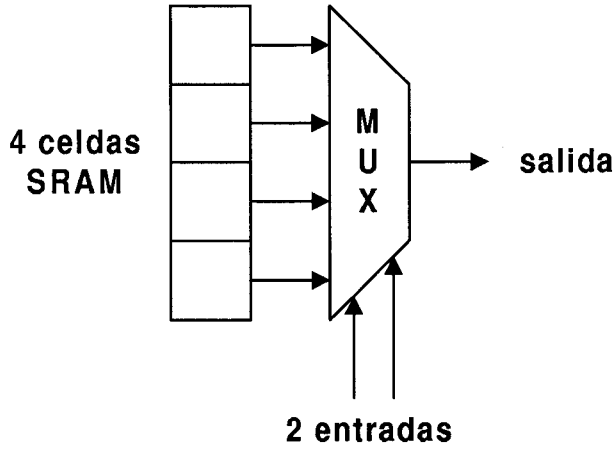


Figura 1.2: Esquema de una LUT de 2 entradas

gran mayoría de las FPGAs modernas usan bloques lógicos compuestos de más de una 4-LUT interconectadas con algunos biestables. Si bien el uso de 4-LUTs, es muy común, el número óptimo de éstas en el interior del bloque lógico no está claro. Por ejemplo, la serie 3000 de Xilinx [XIL99], contiene 2 4-LUTs, mientras que la serie 4000 contiene 2 4-LUTs y una 3-LUT interna que permite combinar las otras dos. Existen muchos artículos publicados al respecto; en [HE93] Jianshe He y Rose, estudiaron FPGAs que contenían LUTs de dos tamaños distintos, llegando a la conclusión de que la FPGA con máxima eficiencia en área debía poseer bloques lógicos formados por 2 LUTs de 4 entradas y una LUT de 3 entradas. Otros estudios como el que aparece en [KAR91], muestran que un bloque lógico con 2 4-LUTs presenta la mayor eficiencia en área, mientras que en [HIL93] se describe un bloque lógico formado por 4 LUTs de 3 entradas como óptimo en área y uno formado por 2 4-LUTs como óptimo en velocidad.

Salvo estas discrepancias, debido en gran medida a los diferentes programas utilizados para realizar las pruebas y a las restricciones impuestas en cada estudio, en general es muy común [BET99b] considerar

un bloque lógico compuesto por uno o varios elementos lógicos básicos (ELB), como se presenta en la figura 1.3, donde cada ELB está formado por una k-LUT y un biestable (ver figura 1.4). Tal es el caso de la FPGA Virtex que combina cuatro de estos ELBs para formar su bloque lógico.

El mayor problema que presentan los bloques lógicos complejos (de más de un ELB) es la gran cantidad de conexiones necesarias para interconectarlos. Por ejemplo, el bloque lógico de la Virtex, que posee 4 LUTs, presenta 16 entradas que deberán ser conectadas a alguna salida. En el capítulo 2 se presenta un estudio sobre las ventajas de compartir entradas entre LUTs, analizando la eficiencia en área resultante del dispositivo y la disminución de la conectividad.

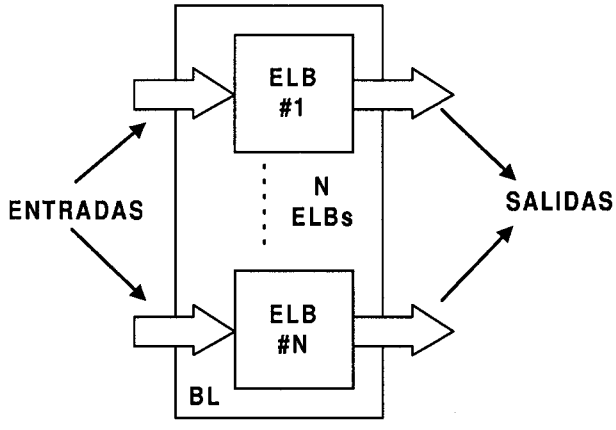


Figura 1.3: Esquema general de un bloque lógico

1.1.2 Recursos de rutado

La descripción de la arquitectura de conexionado es de gran importancia, y presenta un compromiso entre las necesidades de conexionado, dependientes en gran medida de la arquitectura del bloque lógico usado, y el área necesaria para realizarla.

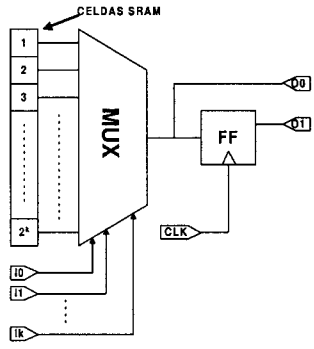


Figura 1.4: Elemento lógico básico (ELB)

Existen tres grandes tipos de FPGAs según su arquitectura de rutado [BET99b]:

- *tipo-isla*: Como las de Xilinx[XIL99] o Lucent[LUC00], donde los canales de rutado rodean al bloque lógico.
- *tipo-fila*: Como las de Actel[ACT99a], donde los bloques lógicos se estructuran en filas en modo similar a un Gate-Array y reserva los espacios entre ellas para rutado.
- *jerárquica*: Como las de Altera[ALT99], donde los bloques lógicos se agrupan en conjuntos que poseen conexiones locales, y a su vez dichos conjuntos se interconectan mediante estructuras de conexión superiores.

Esta tesis sólo estudia las FPGAs *tipo-isla* por lo que a continuación se procederá a describir su arquitectura.

En la figura 1.5 puede observarse una arquitectura de este tipo. Los bloques lógicos (BL), están rodeados por canales de rutado formados por segmentos de metal. Las entradas o salidas de estos bloques lógicos, que llamaremos *pines*, pueden conectarse a un cierto número de estos

segmentos a través de la denominada *matriz de conexión* , en adelante *bloque-C* [ROS91], formada por interruptores programables. En cada cruce entre canales horizontales y canales verticales, existe una matriz de interruptores o bloque-S [ROS91], cada uno de los cuales contiene una serie de interruptores programables que permiten conectar segmentos de un canal con los siguientes del mismo canal o con otros segmentos de otro canal, consiguiendo así prolongar las conexiones a lo largo del dispositivo.

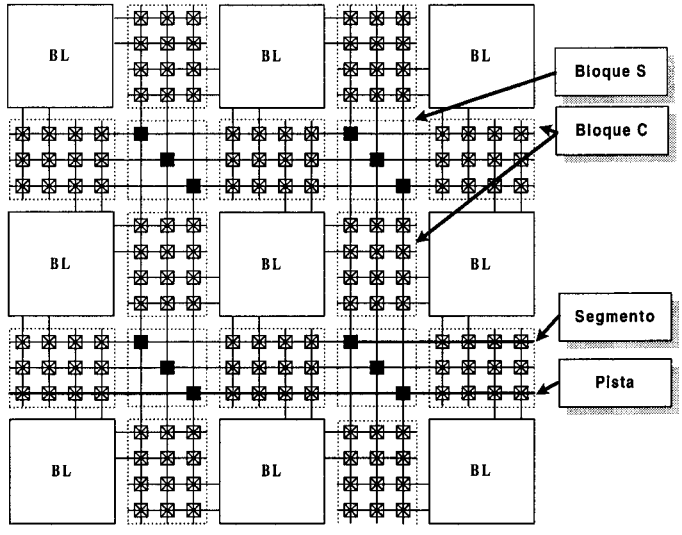


Figura 1.5: Estructura de rutado FPGA *tipo-isla*

A partir de la publicación de [ROS91], la mayoría de los estudios realizados sobre este tipo de FPGAs usan la notación que aparece en el artículo, muy adecuada para describir los parámetros básicos de esta arquitectura. Se usa la letra W para referirse al número de pistas contenidas en un canal de rutado. El número de segmentos a los que puede ser conectado un pin de un bloque lógico se denomina *flexibilidad de la matriz de conexión* o F_c. El número de segmentos a los que puede conectarse otro segmento en un bloque-S se denomina *flexibilidad de la*

matriz de interruptores o F_s. Por ejemplo, en la figura 1.5, $W = 3$, $F_c = 3$ y $F_s = 3$.

La mayoría de los estudios publicados hasta ahora, han considerado arquitecturas de rutado compuestas por segmentos de longitud unidad, es decir, que sólo pasan por un bloque lógico. En [ROS91] Rose y Brown estudiaron este tipo de FPGAs y encontraron que el valor óptimo de los parámetros anteriores (F_c y F_s) es de $F_s = 3$ ó 4 y F_c entre $0.7W$ y $0.9W$. Sin embargo, también puede obtenerse una buena rutabilidad con una $F_s = 2$ si los algoritmos empleados en el rutado están especialmente optimizados para este tipo de arquitectura [TSE92].

Típicamente, las FPGAs tipo SRAM comerciales, utilizan multiplexores para la configuración de las conexiones de los pines de entrada de los bloques lógicos. Esta técnica reduce en gran medida la cantidad de memoria utilizada para la configuración de dichos pines, pero por otra parte impide la conexión de estos pines a varios segmentos de rutado al mismo tiempo. En el caso de los pines de entrada no presenta ningún problema, ya que generalmente sólo llega una señal a estos pines. No ocurre así para los pines de salida, los cuales suelen estar conectados a más de una entrada. Si bien esta técnica reduce la rutabilidad, en el capítulo 4 se presenta un estudio sobre el ahorro en área de silicio producido por la utilización de decodificadores para la conexión de los pines de salida de los bloques lógicos y su repercusión en el retraso y en la rutabilidad de la FPGA.

También se presenta en el capítulo 4, un modelo de área para FPGAs en función de los parámetros básicos vistos anteriormente, lo que permite una estimación del área ocupada por el dispositivo en la etapa previa al

diseño.

1.2 Herramientas de síntesis automática para FPGAs

Las herramientas de síntesis automática permiten generar la configuración de cada uno de los elementos programables de una FPGA a partir de la descripción de un circuito en un lenguaje de alto nivel como *VHDL* o *VERILOG*. El usuario posee normalmente dos posibilidades a la hora del diseño, la primera consiste en realizar directamente la descripción del circuito mediante uno de estos lenguajes. Otra posibilidad consiste en generar esa descripción mediante una captura de esquemas usando bloques de librería previamente definidos por el fabricante. Independientemente de la opción utilizada, el resultado es una descripción del circuito en lenguaje de alto nivel.

El proceso de convertir la descripción de un circuito en lenguaje de alto nivel a una lista de bits que configura por completo la FPGA, es complicado por lo que suele dividirse en varias etapas. En la figura 1.6 puede verse el diagrama de flujo típico de estas herramientas. Cada uno de estos procesos se describe en los apartados siguientes.

1.2.1 Síntesis

La primera etapa consiste en convertir la descripción del circuito en una lista (*netlist*) de puertas lógicas básicas, que a su vez serán empaquetadas en bloques lógicos de la FPGA. Este proceso es tan complejo que normalmente suele dividirse en dos o más etapas:

1. Optimización lógica

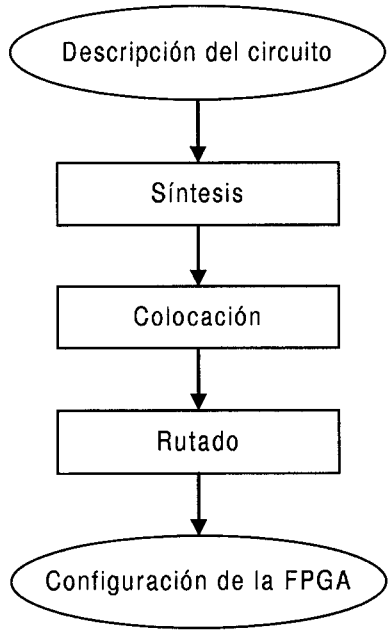


Figura 1.6: Diagrama de flujo

2. Generación de las funciones lógicas en forma de LUTs, *"Technology Mapping (TM)"*

3. Empaquetado en bloques lógicos

La primera fase es independiente del dispositivo utilizado, y consiste en eliminar la lógica redundante y simplificarla cuanto sea posible. La lógica resultante es entonces mapeada en LUTs en la segunda fase. Estas dos etapas han sido objeto de muchos estudios [KAR91] [FRA91] [HWA94] [FAR94], incluso existen herramientas de libre distribución como [SEN92] y [FRA91]. Por este motivo no se profundizará más en este tema.

El último paso es necesario si el bloque lógico de la FPGA contiene más de una LUT. La fase de empaquetado agrupa varias LUTs y biestables en un bloque lógico, respetando las limitaciones impuestas por el número de LUT máximo que un bloque lógico puede contener y el

número máximo de señales de entrada y relojes del bloque. El algoritmo de empaquetado persigue dos objetivos:

1. minimizar el número de bloques lógicos utilizados
2. empaquetar LUTs interconectadas en el mismo bloque lógico, para minimizar así el número de interconexiones entre distintos bloques.

Este es el clásico problema de particionamiento o clustering, pero al existir varias restricciones impuestas por la propia arquitectura del bloque lógico, estos algoritmos se muestran ineficaces frente a algoritmos heurísticos de búsqueda secuencial [BET99b]. Por este motivo, no existen muchas publicaciones sobre este tema, ya que dependen en exceso de la arquitectura usada. Entre otros, en [CON96c], se presenta un algoritmo aplicable a varios tipos de bloque lógico, mientras que en [BET97b] o [BET99b] se desarrolla un método de empaquetado para una estructura concreta.

En el capítulo 2, se presenta una forma rápida y sencilla de empaquetado para bloques lógicos complejos, donde las LUTs comparten un cierto número de entradas, y donde el criterio de optimización se basa en la minimización de la interconectividad del circuito.

1.2.2 Colocación

El siguiente paso en cualquier herramienta automática es el de la colocación (Placement). La etapa anterior genera una lista de bloques lógicos y ahora se debe de elegir su emplazamiento dentro de la FPGA. En este proceso se busca optimizar los siguientes puntos:

1. minimizar el retraso

2. maximizar la rutabilidad

A grandes rasgos podemos dividir los algoritmos existentes en dos grandes familias: los que utilizan la técnica de enfriamiento simulado y los que usan métodos de particionamiento de corte mínimo *"min-cut"*. En la primera se intenta reproducir el proceso de enfriamiento que se aplica al metal en estado líquido cuando se quieren conseguir objetos metálicos de gran regularidad en su estructura cristalina [KIR83]. La segunda consiste en dividir el circuito en pequeños subconjuntos tratando de minimizar la interconexión de estos conjuntos, consiguiendo así minimizar la congestión de los canales de rutado de la FPGA [KLE91]. La generalidad de la técnica de enfriamiento simulado la hace aplicable a cualquier problema de optimización, con lo que resulta mucho más sencillo añadir un nuevo objetivo o una nueva restricción al algoritmo. Por este motivo, sólo estudiaremos los algoritmos basados en enfriamiento simulado.

Estos algoritmos son aplicables a diversas funciones de coste, de entre las cuales caben destacar las de corte mínimo y las de distancia o retraso. Las primeras tratan de minimizar la congestión de los canales de rutado mientras que las segundas minimizan la distancia entre bloques lógicos conectados, lo cual es equivalente a reducir el retraso de dichas conexiones (es de suponer que al tener que recorrer menos distancia, una conexión atravesará menos interruptores programables, disminuyendo así su retardo). La eficiencia de estas funciones de coste, ha sido sobradamente probada para FPGAs con estructuras simétricas. No ha sido así para FPGAs con un elevado grado de asimetría, una FPGA rectangular o con recursos de rutado diferentes para canales de rutado horizontales y verticales, presenta serias dificultades para estas funciones de coste. Principalmente, como veremos en el capítulo 3, debido a que éstas no tienen en cuenta la asimetría. En los SOCs, la existencia

de estructuras de conexionado especiales que permiten la conexión de la FPGA con los bloques adyacentes (convertidores, amplificadores, microprocesador,...) aumenta en gran medida la asimetría del dispositivo.

En el capítulo 3 se presenta un nuevo algoritmo basado en enfriamiento simulado, que permite tomar en cuenta el retraso, la congestión de los canales de rutado y cualquier tipo de asimetría del dispositivo, además de ofrecer el mejor comportamiento frente a arquitecturas reales.

1.2.3 Rutado

Una vez que los bloques lógicos han sido colocados, el rutado determina qué interruptores programables deben estar cerrados para conectar los pines de entrada y de salida de los bloques lógicos. Generalmente se representan los recursos de rutado de la FPGA mediante un grafo dirigido [NAG98], cada segmento es representado mediante vértices y cada posible conexión mediante una arista. Se utiliza un grafo dirigido debido a la posible existencia de elementos que implican una dirección como multiplexores o búfferes triestado.

Rutar una conexión significa encontrar un camino entre los vértices que representan a los pines de los bloques lógicos que deben ser conectados. Algunos criterios de optimización son los siguientes:

- minimizar el número de segmentos utilizados, ya que los recursos de rutado suelen ser escasos.
- minimizar el retraso introducido por cada conexión, este criterio es importante sobre todo en las conexiones que pertenecen al camino crítico del circuito.

- impedir que varias conexiones usen los mismos recursos de rutado

Existen dos formas de atacar este problema, en una sola etapa como en [WU97] y [BET99b], o dividiendo el proceso en dos etapas, una primera llamada *rutado global* que elige los pines de los bloques lógicos y los canales de rutado por los que va ir cada conexión [THA97], y una segunda, el *rutado detallado*, que se encarga de seleccionar los segmentos que la conexión va a usar dentro de los canales de rutado seleccionados por el rutado global [BRO92c]. El rutado detallado es un proceso muy complicado que conlleva el riesgo de no encontrar una solución debido a la limitada flexibilidad de los recursos de rutado y a las restricciones impuestas por el encaminador global. Por este motivo suele recomendarse, como en [BET99b], la resolución del problema en una sola etapa. Sin embargo, en FPGAs asimétricas o SOCs, el uso de dos etapas puede ser de gran ayuda, ya que permite de alguna manera modelar de forma más eficiente las irregularidades de la estructura.

En el capítulo 4 se presenta un algoritmo de rutado detallado especialmente diseñado para permitir el uso de decodificadores en los pines de salida de los bloques lógicos.

Capítulo 2

Empaquetado

En el capítulo anterior se realizó una introducción a la tercera etapa del proceso de síntesis automática, el empaquetado. Mientras que la optimización lógica y la generación de las funciones lógicas en forma de LUTs pueden realizarse de forma totalmente independiente de la FPGA utilizada, el empaquetado sí requiere de un conocimiento de la arquitectura de los bloques lógicos. Este capítulo presenta un estudio sobre arquitecturas de última generación, como las nuevas familias VIRTEX de Xilinx [XIL99] y FIPSOC de SIDSA (Semiconductores Investigación y Desarrollo S.A.) [SID99a], demostrando la posibilidad de incrementar la rutabilidad de los circuitos sin aumentar el área de silicio ocupada por el dispositivo.

2.1 Introducción

Como ya se ha visto en el capítulo anterior, la eficiencia en área de una FPGA viene determinada en gran medida por la granularidad y estructura de sus bloques lógicos. Una FPGA formada por bloques lógicos simples (también llamada de grano fino), necesita de muchos de ellos para implementar un diseño, y aunque la unidad lógica se utilizará sin

desperdicio de su capacidad, el área de rutado necesitada para interconectar dichos bloques lógicos será muy alta y normalmente poco aprovechada. Por el contrario, si se usan bloques lógicos muy complejos (FPGA de grano grueso), podrán ser empaquetadas funciones muy complejas en ellos, aunque por lo común gran parte de la funcionalidad disponible no será utilizada, o lo que es lo mismo, gran parte del área de la FPGA podría ser desperdiciada.

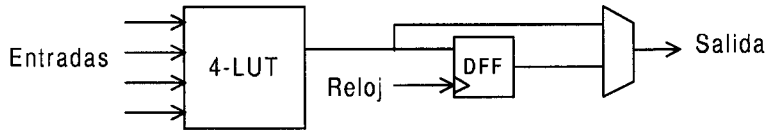
Recientemente, han aparecido nuevas familias de dispositivos programables basados en tecnología SRAM, como la nueva VIRTEX de Xilinx [XIL99], la APEX de Altera [ALT99] y la familia FIPSOC de SIDSA [SID99a]. Este último caso va más allá del concepto de FPGA, ya que posee una parte analógica programable, un microprocesador y una FPGA, todo en un solo circuito integrado, lo que se ha llamado *un sistema en un chip*.

La FPGA interna de FIPSOC y las mencionadas anteriormente, poseen una característica común, los bloques lógicos que utilizan son de grano grueso. Básicamente, FIPSOC posee bloques lógicos formados por cuatro LUTs de cuatro entradas, cuatro biestables y una lógica adicional interna de extraordinaria utilidad, la cual permite la configuración de los bloques en modo *MACRO*, es decir, en sumadores, contadores, multiplicadores, y otros. Ésta es precisamente una de las grandes ventajas de los bloques lógicos de grano grueso: muchas de las interconexiones entre la lógica, se realizan en el interior de los bloques lógicos, sin utilizar recursos de rutado externos, disminuyendo los retrasos (las conexiones internas presentan menores retrasos) y posibilitando una frecuencia de funcionamiento mucho mayor. Sin embargo, como ya se ha dicho anteriormente, la gran capacidad de estos bloques lógicos es desperdiciada

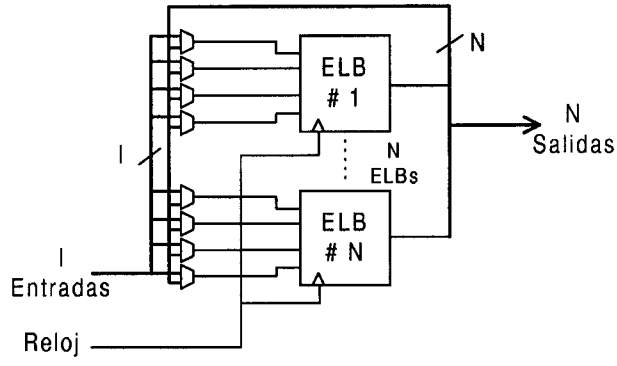
en muchos diseños, reduciendo la eficiencia en área del dispositivo.

Esta reducción de la interconectividad (medida del número de conexiones entre los bloques lógicos de un circuito) puede mejorarse. En [HIL93] *Hill y Woo* presentan un estudio sobre cómo al compartir un cierto número de entradas entre LUTs de un mismo bloque lógico, puede aumentarse la eficiencia en área. Al compartir entradas, el número de celdas SRAM necesarias para configurar las conexiones disminuye, con lo cual el área de silicio también. Sin embargo, los resultados presentados no son de mucha ayuda ya que el autor sólo estudia los casos extremos, es decir, por una parte estudia el efecto en la eficiencia en área de bloques lógicos formados por LUTs que no comparten ninguna entrada y posteriormente aquellos que comparten todas sus entradas. Como es de esperar, el peor caso de eficiencia en área es para aquellos bloques lógicos formados por LUTs que comparten todas sus entradas, ya que la fuerte restricción impuesta hace que la mayoría de ellos estén parcialmente ocupados, dando lugar a un gran desperdicio de área. Una consecuencia importante que los autores no tienen en cuenta es la reducción de la interconectividad antes mencionada, lo cual repercute favorablemente en la rutabilidad de los diseños, y debe ser por tanto sometida a estudio.

En [BET97b], se presenta un bloque lógico (ver figura 2.1) formado por elementos lógicos básicos, y una serie de multiplexores internos que permiten realimentaciones en el interior del bloque. Los autores también tienen en cuenta la posibilidad de compartir entradas entre LUTs de un mismo bloque lógico, para ello usan de nuevo multiplexores para seleccionar las cuatro entradas de las LUTs de entre las I entradas posibles. El artículo estudia la eficiencia en área de una FPGA formada por estos



(a) Elemento lógico básico (BLE)



(b) Bloque lógico

Figura 2.1: Estructura de un bloque lógico

bloques lógicos y el valor óptimo de I en función del número de elementos lógicos por bloque lógico. En el caso de las VIRTEX formadas por 4 elementos básicos, los autores señalan que con $I = 10$ se obtiene la mayor eficiencia en área. Es decir, sólo son necesarias 10 entradas (de las 16 existentes) accesibles desde fuera del bloque para conseguir la mejor eficiencia. El mayor problema presentado por esta arquitectura es el área consumida por los multiplexores y sus correspondientes bits de configuración. Para el caso anterior, son necesarios 16 multiplexores de 10x1 y 64 celdas SRAM para seleccionar cada una de las entradas de las LUTs, un área considerable teniendo en cuenta que la mayoría de las arquitecturas modernas utilizan multiplexores para seleccionar el segmento de conexión del canal de rutado adyacente, con lo que el aumento en área del bloque lógico puede llegar a ser muy importante. Sin embargo, el número de interconexiones se reduce considerablemente, facilitando las tareas posteriores de colocación y rutado.

Podría pensarse que otro problema es el retraso introducido por los multiplexores. Los resultados presentados en [AHM00], demuestran que el uso de bloques lógicos complejos, de más de un ELB, reducen de forma significativa los retrasos introducidos por el rutado de las conexiones. Por este motivo, el aumento del retraso interno de los bloques lógicos, debido a los multiplexores, es insignificante frente a la disminución del retraso del rutado.

2.2 Nueva arquitectura de bloque lógico

Los resultados de los artículos anteriores son prometedores, parece muy probable encontrar una estructura para un bloque lógico con un área menor y que aumente la rutabilidad de los diseños sin reducir excesivamente la efectividad en área del dispositivo.

Para la reducción en área puede seguirse la línea apuntada por *Hill y Woo* [HIL93]. Si las LUTs comparten algunas de sus entradas, puede ahorrarse la memoria de configuración de estos pines. Al compartir dichas entradas se reduce el número de conexiones del diseño, aumentando su rutabilidad. Según el estudio de Rose y Betz [BET97b], no es del todo descabellado pensar que en un diseño cualquiera las LUTs compartan entradas, sin embargo también es cierto que al no utilizar multiplexores, el número óptimo de entradas a compartir disminuirá debido al decremento en la flexibilidad del bloque lógico. Queda por saber cuál es el máximo número de entradas que pueden ser compartidas entre las LUTs sin reducir la efectividad en área de la FPGA.

En la figura 2.2 puede verse la arquitectura de la parte combinacional

del bloque lógico bajo estudio. La parte secuencial del mismo se supone igual a la utilizada en el ejemplo anterior. Esta estructura es muy similar a la utilizada en FIPSOC, donde grupos de dos LUTs comparten entradas, aunque en este caso el hecho de compartir entradas responde a unas necesidades particulares de esta arquitectura que nada tienen que ver con la reducción en área del bloque lógico.

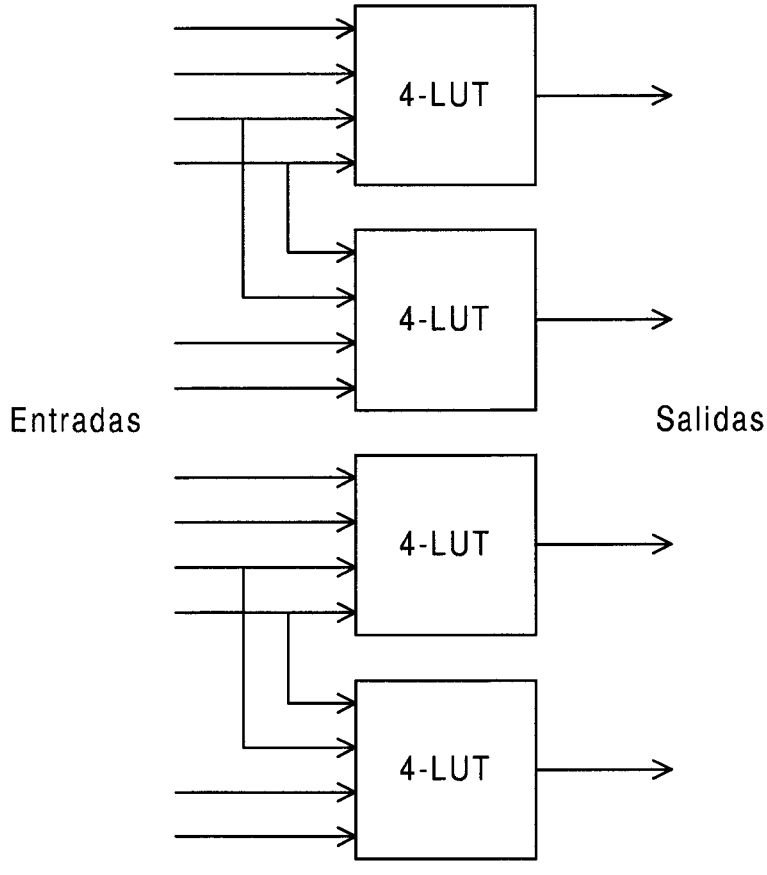


Figura 2.2: Arquitectura de bloque lógico

Si I es el número de entradas compartidas entre dos LUTs, se quiere estudiar el efecto de I sobre:

1. Eficiencia en área del dispositivo.

2. Rutabilidad del circuito.

3. Área consumida por el bloque lógico.

Para ello, se necesitará de un algoritmo capaz de empaquetar las LUTs que conforman el circuito de prueba, teniendo en cuenta el número de entradas compartidas por éstas dentro de un bloque lógico.

2.3 Algoritmo de empaquetado

En primer lugar haremos una simplificación del problema, debido a que el estudio a realizar sólo afecta a la parte combinacional de los bloques lógicos, se utilizarán para las pruebas circuitos puramente combinacionales, los resultados así obtenidos no suponen ninguna pérdida de generalidad.

2.3.1 Formulación

El problema es el siguiente, dado un conjunto de LUTs interconectadas (*circuito de entrada*), el algoritmo debe dividir el circuito en subconjuntos de, como máximo dos LUTs, tal que, el número de entradas compartidas entre LUTs de un mismo conjunto sea mayor o igual a I, siendo I un parámetro externo. La finalidad de dicho algoritmo será:

1. Minimizar el número de subconjuntos creados

2. Maximizar la rutabilidad del circuito

El segundo objetivo no aparece en ningún otro algoritmo conocido, y se basa simplemente en minimizar el número de conexiones entre bloques lógicos. Una conexión, en general, une una salida a una o varias entradas, lo que comúnmente se denomina *red multipunto*. Si dos o más

entradas son del mismo bloque lógico, sería deseable que los segmentos utilizados para realizar dichas conexiones fueran los mismos y así minimizar el uso de los limitados recursos de rutado. Que esto ocurra, depende de la matriz de conexión o bloque C (ver figura 2.3). Se ha visto que el valor de la F_c, el número de segmentos a los que se conecta un pin, tiene un valor recomendable cercano a $0.8W$, siendo W el número de segmentos que pasan por el bloque C. Existe por lo tanto la posibilidad de conexión al mismo pin utilizando el mismo segmento (conexión A en la figura), pero esta posibilidad depende en gran medida de la densidad de conexiones que cruzan dicho bloque; es decir, debido a la limitación en la flexibilidad de dicho bloque C, en circuitos con alta densidad de interconexión, puede ocurrir que este ahorro en recursos de rutado sea imposible (conexión B en la figura).

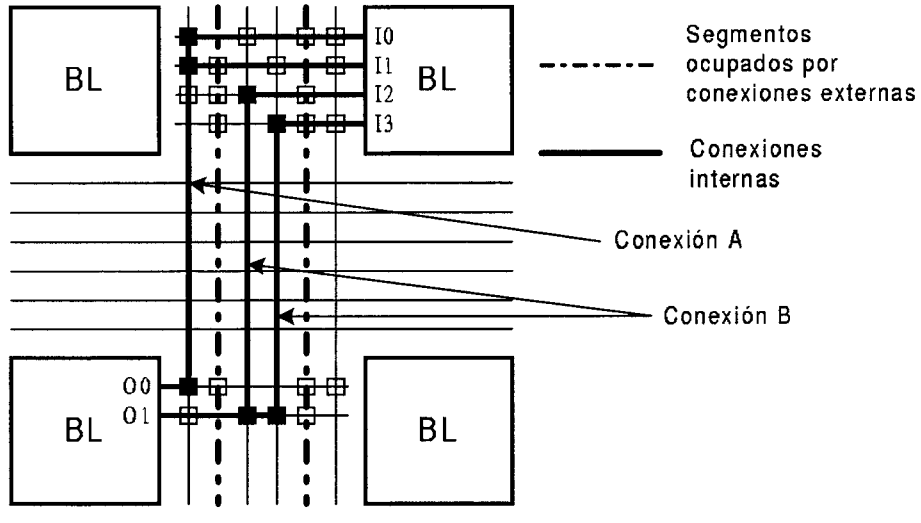


Figura 2.3: Segmentos compartidos

Por este motivo, al compartir las LUTs algunos pines de entrada, se obliga a que sólo se use un segmento para dicha conexión minimizando así el uso de los recursos de rutado y aumentando de alguna manera la rutabilidad del circuito. Sin embargo, no existe una manera clara de

cuantificar dicha mejora en este paso del entorno de desarrollo, lo único realmente cierto es que reduciendo el número de terminales de una red multipunto, reducimos el consumo en recursos de rutado lo cual es beneficioso para las etapas posteriores.

2.3.2 Algoritmo

Para cumplir los objetivos anteriores, se propone el uso de un algoritmo sencillo (ver figura 2.4). Inicialmente el algoritmo parte de un conjunto de LUTs no marcadas, secuencialmente para cada LUT no marcada se busca una segunda LUT no marcada que comparta el máximo número de entradas con la primera, si el número de entradas compartidas es superior o igual a I (parámetro del algoritmo) se marcan las dos LUTs y se crea un subconjunto que contiene esas dos LUTs. Si por el contrario el número de entradas compartidas es inferior a I, el criterio de búsqueda implica que no existe ninguna LUT no marcada que pueda empaquetarse con la primera, por lo que se marca la primera y se crea un subconjunto con una sola LUT. Se continua así hasta que no quede ninguna LUT sin marcar. Al llegar a este punto, todas las LUTs están marcadas y empaquetadas en conjuntos de como máximo 2 LUTs. El paso restante es trivial ya que no existe ninguna restricción y cualquier par de conjuntos puede ser empaquetado en un bloque lógico, aunque para minimizar el número de bloques lógicos interconectados y así facilitar la tarea de colocación de estos bloques en la FPGA, es deseable empaquetar en un mismo bloque lógico conjuntos que compartan el mayor número de entradas.

En el caso de $I = 0$, es decir, las LUTs no comparten ninguna entrada, el resultado obtenido es óptimo siguiendo el criterio de minimización de

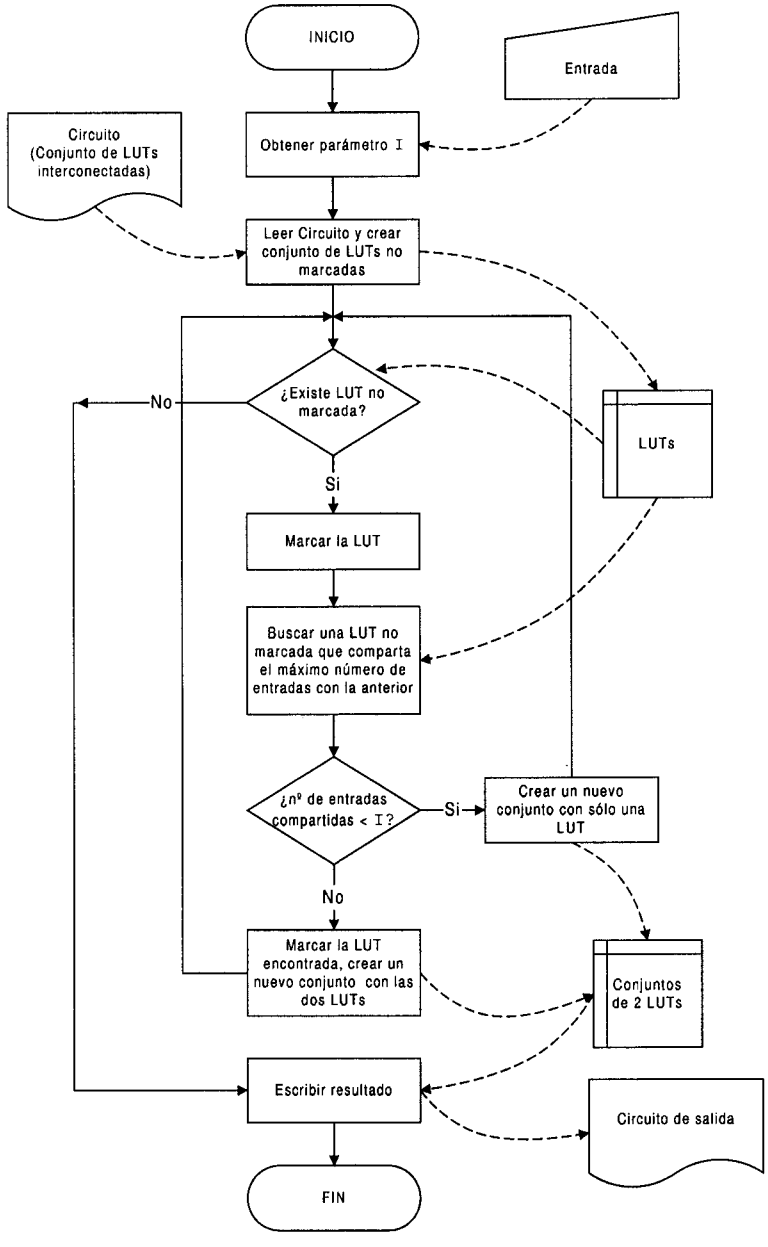


Figura 2.4: Diagrama de flujo

bloques lógicos, de hecho, el número de bloques lógicos obtenidos resulta de dividir por cuatro el número de LUTs del circuito y redondear al entero inmediatamente superior. Para otros valores de I, el algoritmo no es óptimo, pero los resultados obtenidos demuestran, como veremos a continuación, que incluso con este sencillo método pueden conseguirse buenos resultados.

2.4 Resultados

Los resultados que se muestran a continuación, han sido obtenidos usando 10 circuitos de un banco de circuitos de prueba desarrollado específicamente para testear el funcionamiento de algoritmos de síntesis y empaquetado [MCNC93]. Se compone de una serie de circuitos con características específicas, que en su conjunto representa una mayoría de las arquitecturas con las que normalmente tendrá que tratar el algoritmo puesto a prueba. En la tabla 2.1 puede observarse algunas de sus características, como el número de LUTs de 4 entradas y el número de conexiones entre dichas LUTs.

Circuito	Nº 4-LUT	Nº Conexiones
CLIP	75	393
MISEX3C	122	679
PDC	134	791
EX5P	222	1206
C6288	315	3196
ALU4	703	3965
APEX2	720	3526
EX1010	899	4358
SPLA	940	4601
SEQ	1197	6714

Tabla 2.1: Características de los circuitos utilizados

Para cada uno de los circuitos de este banco de pruebas, se ha aplicado el algoritmo para valores de I entre 0 y 4, los resultados así obtenidos pueden verse en las figuras 2.5 y 2.6. Como referencia se ha tomado el resultado de aplicar el algoritmo con $I = 0$, es decir, el resultado de empaquetar las LUTs en bloques lógicos donde las LUTs no comparten entradas. En la figura 2.5 puede verse la disminución del número de redes punto a punto frente al parámetro I (el número de entradas compartidas). En la figura 2.6 se muestra el aumento en el número de bloques lógicos frente al mismo parámetro I.

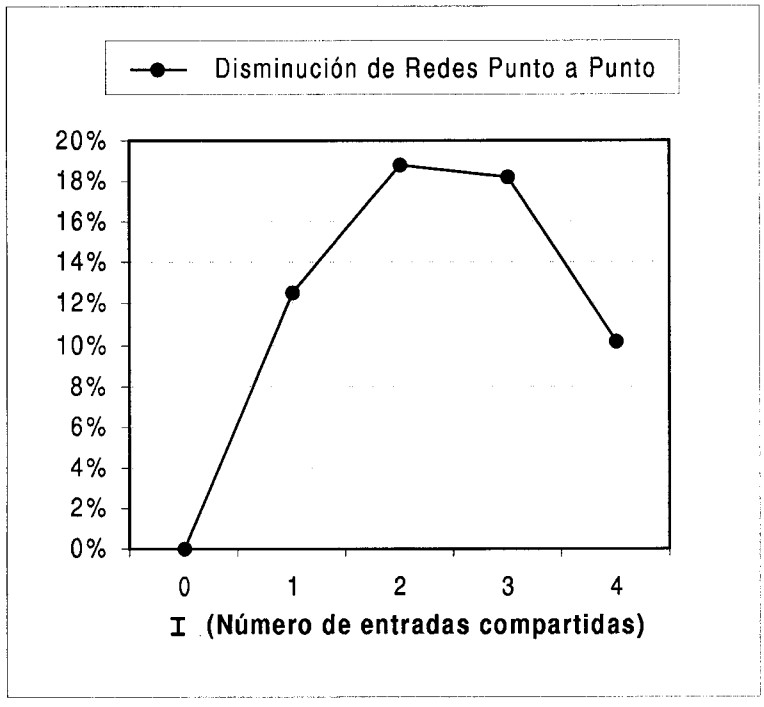


Figura 2.5: Disminución de redes punto a punto

De estas figuras, puede deducirse que el óptimo en cuanto a reducción de redes monopunto se produce cuando $I = 2$ alcanzándose un 19%. En dicha arquitectura, los recursos de rutado serán utilizados de forma más eficiente, mejorando la rutabilidad del circuito. Sin embargo, el precio

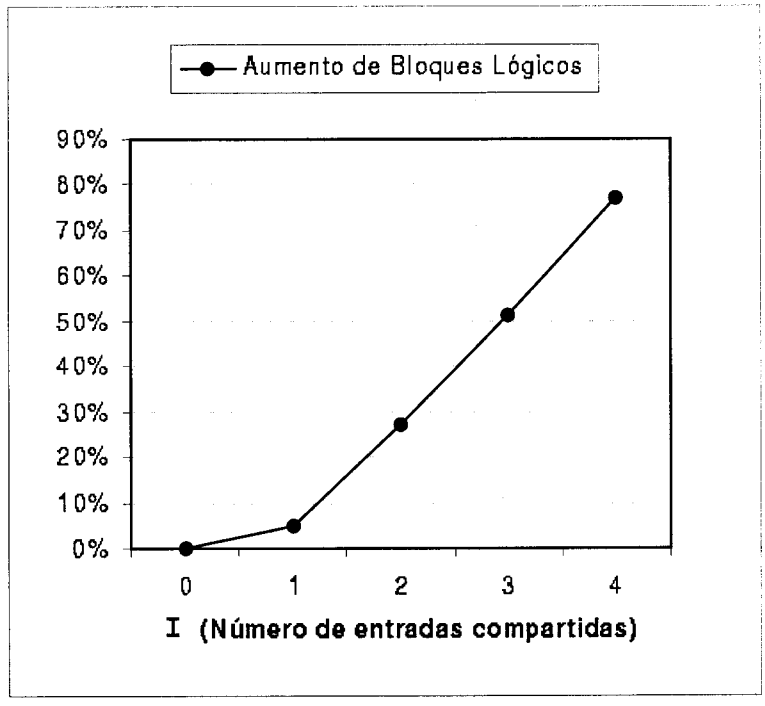


Figura 2.6: Aumento de bloques lógicos

a pagar por compartir dichas entradas es el aumento en un 27% en el consumo de bloques lógicos debido al incremento de bloques lógicos parcialmente ocupados, lo cual se traduce en una menor eficiencia en área. El porcentaje anterior es muy pesimista debido a que los circuitos del banco de prueba han sido generados por una herramienta de síntesis automática que sólo trabaja con LUTs y no emplea los modos *MACRO* de los bloques lógicos, por lo que cabe esperar un mejor comportamiento en la realidad.

Los resultados anteriores no nos dan una idea del posible incremento o decremento en área de silicio, un parámetro muy importante (si no el más importante por su incidencia económica) en el diseño de circuitos integrados. En [DEH96], André DeHon señala que el área de silicio ocupada por un bloque lógico viene determinada por las celdas de

memoria que lo configuran, este hecho permite obtener una aproximación del área consumida por un bloque lógico en función de sus bits de configuración. Sin embargo, esta aproximación sólo es posible si se particulariza para una arquitectura determinada, con la consecuencia de pérdida de generalidad en los resultados. Tomando como ejemplo una simplificación de la arquitectura de FIPSOC [SID99a], el bloque lógico y los recursos de rutado adyacentes son configurados mediante 128 bits, con un total de 192 bits teniendo en cuenta los 64 bits necesarios para configurar las 4 LUTs. Las entradas a dichas LUTs se conectan a los segmentos de los canales de rutado mediante multiplexores de 32x1, por lo que cada entrada utiliza 5 celdas SRAM de las 192 existentes. Ya se ha dicho anteriormente que las LUTs de FIPSOC comparten dos entradas, si no fuera así, el número de celdas de memoria necesarias para configurar un bloque lógico sería de 212 (192 + 4 entradas de más x 5 bits por cada entrada). Dependiendo del valor del parámetro I, el bloque lógico ocupará más o menos área, lo que nos permitirá dar una aproximación del área ocupada por los distintos circuitos del banco de prueba tras pasar por el algoritmo expuesto en el apartado anterior. Dichos resultados en el caso de FIPSOC, pueden verse en la figura 2.7, donde se observa que para $I = 1$ el área no se ve incrementada.

De todo lo dicho en este apartado, cabe destacar una consecuencia de gran importancia: el compartir entradas entre LUTs de un mismo bloque lógico facilita el rutado del circuito, y dependiendo de la arquitectura empleada, el área de silicio ocupada por el diseño puede no verse afectada. En el caso de FIPSOC, para $I = 1$ el área de silicio no aumenta y el número de conexiones punto a punto disminuye en un 13%. Para $I = 2$, el número de redes disminuye en un 19% pero el área aumenta cerca de un 15%. Cabe destacar que los resultados son muy pesimis-

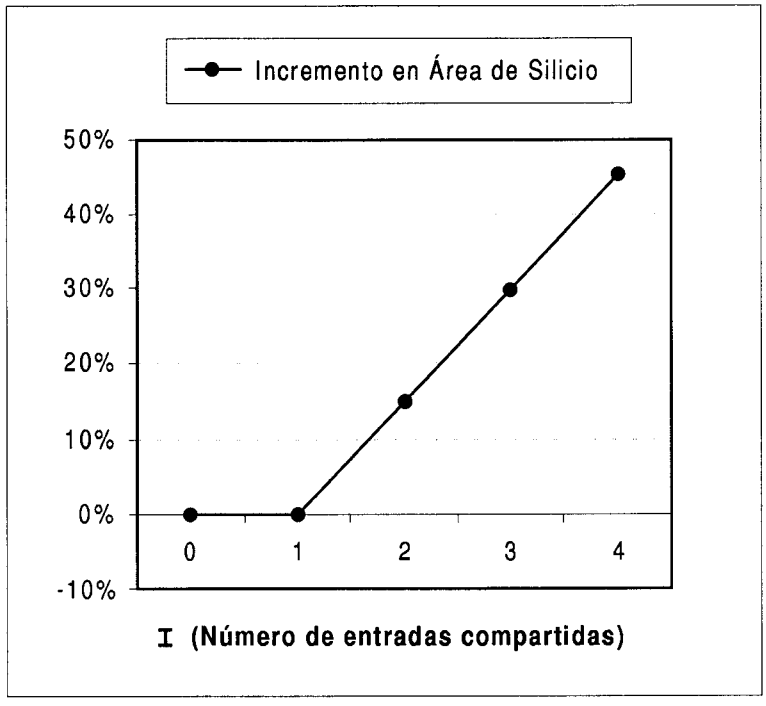


Figura 2.7: Incremento en el área de silicio

tas por 2 razones fundamentales, la primera es la no optimalidad del algoritmo empleado, la segunda es que no se han usado los modos *MACRO* típicos de bloques lógicos complejos que reducen en gran medida el número de conexiones y además facilitan el empaquetado. Por ello, aunque los resultados demuestran que la técnica propuesta es viable, la utilización de MACROs y la particularización del algoritmo a una arquitectura determinada promete aún mejores resultados.

Capítulo 3

Colocación

En este capítulo se presenta la etapa de colocación, con un nuevo algoritmo que presenta el mejor comportamiento en FPGAs reales, es decir, FPGAs con canales de rutado formados por segmentos de diversas longitudes y que además permite la definición de asimetrías, característica muy común en los SOCs (sistemas en un chip). Esta última propiedad, se deriva de la existencia de bloques adicionales como convertidores, amplificadores, filtros, microprocesadores, etc... a los cuales puede conectarse la lógica programable. Dichos bloques rompen la simetría de la FPGA, complicando sobremanera la colocación y el rutado de la misma.

3.1 Introducción

La asimetría de una FPGA puede venir dada por varias razones, si el diseño físico o layout del elemento fundamental de ésta, no es del todo cuadrado, una matriz $N \times N$ de estos elementos puede resultar en un circuito integrado con forma rectangular. Por esta razón, se suele cambiar la relación de aspecto de la FPGA, utilizando una matriz rectangular de bloques lógicos de forma que el diseño físico del circuito

resulte cuadrado. En la figura 3.1 se muestra una FPGA rectangular, en este caso, al existir un mayor número de bloques lógicos horizontales, el número de conexiones en esa dirección será normalmente superior a las existentes en sentido vertical, aumentando la congestión de los canales de rutado horizontales. Para reducir este efecto, puede optarse por aumentar el número de recursos de rutado horizontales, cambiando de nuevo la relación de aspecto del bloque lógico y por lo tanto de la FPGA completa. Esto dificulta el diseño del dispositivo y los algoritmos de las herramientas de colocación y rutado, que hasta ahora han utilizado implícitamente las ventajas de la simetría. Por estos motivos se prefiere un bloque lógico cuadrado, dando lugar a una FPGA cuadrada.

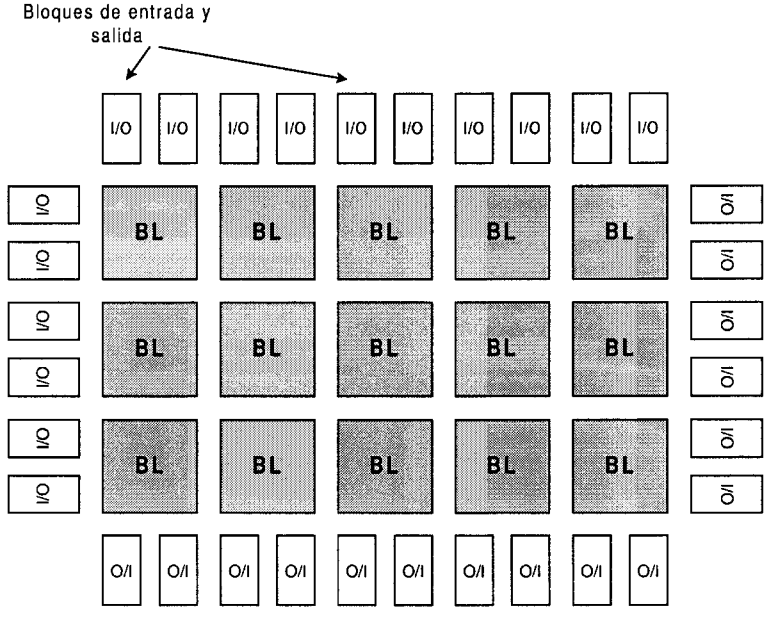


Figura 3.1: Fpga rectangular

Otra causa de asimetría, puede venir impuesta por el propio diseñador, una idea muy común pero errónea de algunos diseñadores es la creencia de que en en una FPGA tipo isla cuadrada, la mayor congestión de los canales de rutado aparece en el centro de la misma, por lo que se

incrementan el número de recursos de rutado en la región central de la FPGA, complicando sobremanera el diseño de la misma. Estudios como los de [BET99b] demuestran dicho error, la congestión de los canales de rutado cercanos al centro no es mucho mayor que los de la periferia, por lo que generalmente se debe tender a la uniformidad de los recursos de rutado. Este mismo estudio refleja también la existencia de los denominados *puntos negros*, lugares dentro de la FPGA que presentan una congestión elevada de los canales de rutado, cuya posición depende en gran medida del circuito que se quiere rutar. Esto quiere decir que, dada una arquitectura de rutado y dependiendo del circuito a rutar, existirán unas zonas muy congestionadas que dificultarán el rutado en unas posiciones que variarán según el circuito.

En los SOCs, la asimetría radica en la existencia de bloques adicionales de naturaleza diferente a la FPGA propiamente dicha. Estos bloques, que se comunican con la lógica programable, poseen unas entradas y salidas especiales, situadas en algunos puntos de la FPGA que rompen la simetría del diseño. Además, conseguir una matriz cuadrada de bloques lógicos resulta más complicado debido a la presencia de estos bloques, con lo que no es de extrañar que la FPGA posea un grado de asimetría elevado, como en el caso de FIPSOC [SID99a].

3.1.1 Enfriamiento simulado

La mayoría de los algoritmos de colocación se basan en la técnica de enfriamiento simulado debido a su gran adaptabilidad. Para facilitar la comprensión de lo expuesto en este capítulo, el autor ha creído necesario incluir aquí, y no en un anexo posterior, un breve y simplificado resumen del proceso de enfriamiento simulado. El pseudocódigo de dicho proceso

aplicado a la colocación de los bloques lógicos de la FPGA puede verse a continuación:

```
P = ColocaciónAleatoria();
T = TemperaturaInicial();
mientras (CriterioFinal()==Falso){
   mientras (BucleInterno()==Falso){
      P_nuevo = GenerarMovimiento(P);
      ΔC = Coste(P_nuevo) - Coste(P);
      r = Aleatorio(0,1);
      si (r<e^(-ΔC/T)){
         P = P_nuevo;
      }
   }
   T = ActualizarTemperatura();
}
```

El proceso comienza con una asignación de los bloques lógicos a posiciones aleatorias dentro de la FPGA. A partir de esta solución inicial, para cada bloque seleccionado aleatoriamente, se genera un movimiento también aleatorio. El criterio de aceptación de dicho movimiento depende de la temperatura a la cual se realiza y de una cierta función de coste. A temperatura alta, mayor es la probabilidad de que el movimiento sea aceptado, independientemente de la función de coste debido al termino $e^{-\Delta C/T}$. A medida que desciende la temperatura, la probabilidad de aceptación de movimientos que empeoran el coste disminuye también. La posibilidad de aceptar movimientos con coste alto permite al algoritmo escapar de los llamados mínimos locales de la función de coste.

La tasa de disminución de la temperatura, el criterio de finalización del algoritmo, el número de movimientos intentados para cada valor de

temperatura y el método de generación de estos movimientos aleatorios definen el motor del algoritmo. Un buen motor es crucial para obtener buenos resultados en un tiempo aceptable. En [HUA86], se presenta un motor especialmente diseñado para la colocación de celdas estándar (standard-cells). En la colocación inicial, se generan una serie de movimientos aleatorios, la temperatura inicial del algoritmo se fija a 20σ donde σ es la desviación estándar del coste de los movimientos generados. La actualización de la temperatura se calcula a partir de:

$$T_{nueva} = T_{previa} \cdot e^{\frac{-\lambda T_{previa}}{\sigma}}$$

donde λ es una constante con un valor típico de 0.7 y σ es la desviación estándar de los movimientos aceptados durante T_{previa}. El algoritmo finaliza cuando la diferencia entre el máximo y el mínimo de los costes aceptados a una determinada temperatura, iguala al coste máximo de cualquier movimiento a esa temperatura.

3.1.2 Funciones de coste

El motor del algoritmo es totalmente genérico y no depende de la tarea a la cual se aplica. El problema surge con las funciones de coste, que sí dependen de la aplicación. Básicamente existen dos tipos de funciones de coste:

1. Modelado de retrasos
2. Modelado de congestión

El primero de ellos, se basa en la minimización de los retrasos de las conexiones entre bloques lógicos, también conocida como *Timing Driven*. La más elemental está basada en el cálculo del semiperímetro de las

conexiones y ha sido trasladada directamente del problema de colocación también presente en *celdas estándar*. Dicha función de coste puede verse a continuación:

$$F_c = \Sigma_{redes} \left[(max_x - min_x) \cdot K_x + (max_y - min_y) \cdot K_y \right]$$

donde x, y son las coordenadas de los bloques lógicos conectados por la red, k_x y k_y son constantes que pueden ser sintonizadas en el caso de que el número de líneas de segmentos por canal sea distinto para canales horizontales y verticales. Dicha función de coste procede directamente del problema de colocación en celdas estándar, donde los canales de rutado no poseen una anchura fija y pueden ser aumentados o reducidos a voluntad. Aunque existan modificaciones como las de [EBE95] que toman en cuenta algunas propiedades locales de las redes, representar adecuadamente algún tipo de asimetría mediante esta función de coste es una tarea complicada.

El segundo tipo de función de coste trata de minimizar la congestión de los canales de rutado. Para ello, calcula el número de redes que *cortan* o cruzan una determinada *barrera*, la cual representa un canal de rutado o la división de dos zonas. Minimizando el número de cortes por dichas barreras, se disminuye la congestión de los canales.

En general, los algoritmos basados en modelos de retrasos generan soluciones compactas en cuanto a la distribución de los bloques lógicos, esto conlleva un máximo aprovechamiento de los recursos de rutado en detrimento de la congestión de los canales. Con este tipo de algoritmos, puede suceder que un diseño pequeño en comparación con la FPGA, pero con un alto grado de conectividad, resulte imposible de rutar. Por otra parte, los algoritmos basados en modelos de congestión tienden a

expandir al circuito dentro de la FPGA, tratando de maximizar su rutabilidad a expensas de desaprovechar recursos de rutado e incrementar el retraso en las conexiones.

De estos dos tipos de funciones, la que mejores resultados ha dado en cuanto a retraso, rutabilidad y tiempo de computación, puede verse en [BET97a] y [BET98]. La función de coste consiste en el cálculo del semiperímetro de las redes del circuito, pero además incluye un término que permite caracterizar las posibles diferencias entre el número de segmentos de los canales de rutado. Los resultados presentados por los autores son los mejores para FPGAs de diversas características, en comparación con otros algoritmos publicados. Sin embargo, los distintos experimentos realizados por los autores presentan dos características comunes, la primera es que la arquitectura de rutado utilizada para las pruebas esta formada sólo por segmentos de longitud unidad, algo que no ocurre en FPGAs comerciales, donde segmentos largos son fundamentales para mejorar las prestaciones en velocidad del dispositivo. La segunda, es la utilización de FPGAs de tamaño mínimo para el circuito a colocar, es decir, para cada circuito de prueba se utiliza una FPGA cuadrada con el mínimo número de bloques lógicos, así puede comprobarse el funcionamiento del algoritmo en situaciones de máxima congestión. Esto tampoco ocurre siempre en la realidad, está claro que el comportamiento del algoritmo en dichas situaciones es muy importante pero en muchos casos, el circuito a rutar tendrá un número de bloques lógicos inferior al de la FPGA pero no por ello su conectividad será baja. Como veremos a continuación, el algoritmo desarrollado en este capítulo mejora considerablemente los resultados obtenidos para dichas situaciones, conservando un comportamiento óptimo en las pruebas con FPGAs de dimensiones mínimas al igual que el algoritmo anterior.

En las próximas secciones se desarrollará un nuevo algoritmo de colocación, basado en enfriamiento simulado, capaz de modelar cualquier asimetría, y que además posee las ventajas de los dos tipos de funciones de coste anteriores, es decir, permite conseguir las soluciones compactas de los algoritmos basados en modelos de retrasos y a la vez las soluciones más rutables de los algoritmos basados en modelos de congestión.

3.2 Una nueva función de coste: VGC

El objetivo fundamental de cualquier algoritmo de colocación es facilitar y posibilitar el rutado final del circuito. Como objetivo adicional, es deseable reducir el retraso introducido por las distintas conexiones, disminuyendo la distancia entre los bloques lógicos a conectar. Una vez colocados los bloques lógicos, la etapa siguiente de rutado deberá seleccionar los recursos de rutado necesarios para realizar las distintas conexiones. La función de coste que analizaremos en este apartado, permite estimar la congestión en los distintos puntos de la FPGA.

Una forma de facilitar el rutado del circuito, en la etapa de colocación, es estimar correctamente y minimizar la congestión de los canales de rutado. Para conocer cuántas redes o conexiones pasan por una posición dentro de un canal de rutado, debemos conocer el camino seguido por todas las conexiones, lo cual es algo que será determinado en la etapa siguiente, y por lo tanto, totalmente desconocido en la etapa de colocación. Sin embargo, aunque sabemos que existen múltiples posibilidades para conectar entre sí dos bloques lógicos, el camino que ofrece mayores ventajas es el que puede verse en la figura 3.2 [BAE00], por dos razones:

1. Puede aprovechar al máximo la existencia de segmentos largos.

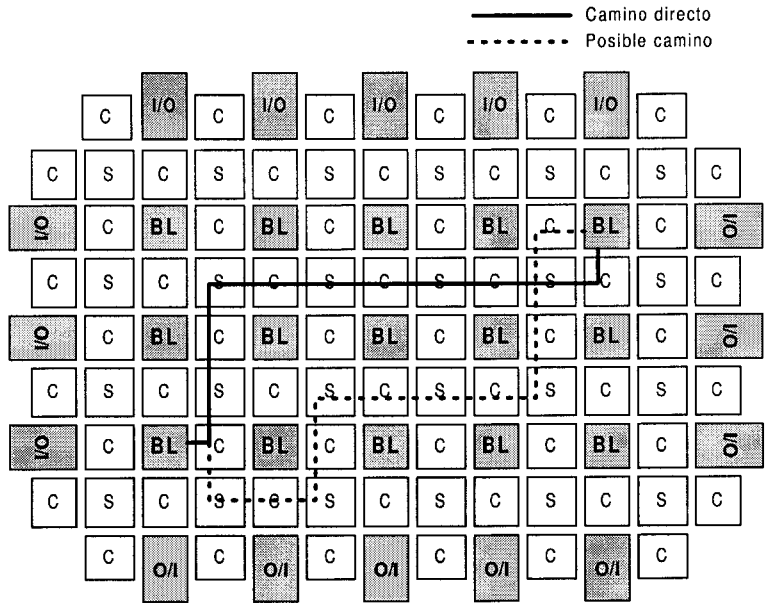


Figura 3.2: Camino directo

2. Al ser el que menos cambios de dirección tiene, pasará normalmente por menos interruptores, por lo que el retraso introducido será menor si existen segmentos largos.

Dicho de otro modo, aunque no sabemos qué camino seguirá la red tras las etapas siguientes, podemos hacer todo lo posible para facilitar el uso de este tipo de caminos.

El *camino directo* de una red es muy sencillo de calcular si se conoce la posición inicial y final de la conexión. Pero en el bucle del algoritmo de enfriamiento simulado, se realizan miles de movimientos que pueden ser o no aceptados. Cada uno de estos movimientos implica el cálculo de dicho camino para cada red que termina o comienza en el bloque movido. Por lo tanto el procedimiento de cálculo debe ser cuidado en extremo para no resultar en un algoritmo demasiado lento.

Suponiendo una FPGA formada por una matriz de $N \times M$ bloques ló-

gicos, existirán por lo tanto $M + 1$ canales verticales y $N + 1$ canales horizontales. Cada canal vertical estará compuesto de $N + 2$ bloques C, y cada canal horizontal estará compuesto de $M + 2$ bloques C.

La fase de inicialización del algoritmo consistirá en la creación de dos matrices de enteros, una de dimensión $(M + 1) \times (N + 2)$ para los bloques C de los canales verticales, y una de dimensión $(N + 1) \times (M + 2)$ para los bloques C de los canales horizontales. Dada la posición inicial del algoritmo de colocación, para cada red o conexión del circuito se calculará el camino directo, incrementándose la congestión de los bloques C por los que pasa dicha red. Una vez determinado el estado inicial del circuito, para cada movimiento generado se actualizará dicha matriz de congestión y se procederá a calcular la función de coste siguiente:

$$\begin{aligned} Coste &= Cross \cdot ((X_{max} - X_{min} + 1) \cdot Med_X[Y_{min}, Y_{max}] \\ &\quad + (Y_{max} - Y_{min} + 1) \cdot Med_Y[X_{min}, X_{max}]) \end{aligned}$$

De la posición de los bloques lógicos, podrán calcularse los valores X_{max}, X_{min}, Y_{max} y Y_{min}, que corresponden a las coordenadas máximas y mínimas, dentro de la FPGA, de los distintos bloques lógicos a los que se conecta la red, esto es, el mismo procedimiento que en el cálculo del semiperímetro. De las matrices de congestión anteriores se obtienen los valores $Med_X[Y_{min}, Y_{max}]$ y $Med_Y[X_{min}, X_{max}]$, el valor medio de la congestión en los canales, horizontales y verticales respectivamente, por los que podría pasar la red, conocidos debido al cálculo anterior del semiperímetro. El parámetro $Cross$ se deriva de un estudio publicado por Chih-Liang [CHE94], donde se demuestra que cuanto mayor es el número de terminales de una conexión, peor es la estimación del semiperímetro mediante el método anterior por lo que es necesario introducir una constante que tenga en cuenta dicho error. El valor de $Cross$

tiene un valor de 1 para redes con 3 o menos terminales y crece lentamente hasta 2,79 para redes con 50 terminales.

La versatilidad de la función de coste queda patente por la posibilidad de definir asimetrías en la arquitectura de rutado, algo difícil de conseguir en los algoritmos de colocación convencionales. Las diferencias entre canales de rutado horizontales y verticales pueden solventarse fácilmente utilizando pesos diferentes para las matrices de bloques C verticales y horizontales anteriormente definidas. Otra posibilidad es la de usar dos vectores de pesos, uno para canales horizontales de dimensión $N+1$ y otro para los verticales de dimensión $M+1$, con lo que se podría tener en cuenta diferencias entre cualquier canal de rutado. Incluso podría definirse una matriz de pesos para tomar en cuenta diferencias entre distintas zonas de la FPGA, pudiendo así, solventar los posibles problemas de congestión de los SOCs en los puntos de conexión con los bloques adicionales típicos de estos dispositivos. Todo esto sin aumentar sensiblemente el tiempo de computación, la definición de dichos vectores se realiza una sola vez en la fase inicial del algoritmo, por lo que el incremento de operaciones se reduce a una multiplicación por una matriz de pesos constantes.

Por último, la propia estructura del algoritmo implica un tiempo elevado de procesamiento lo que elevará seriamente el tiempo de ejecución frente a modelos tan sencillos como el de retrasos, sin embargo, los resultados obtenidos demuestran que dicho incremento permite mejorar considerablemente la rutabilidad de las soluciones obtenidas.

3.3 Experimentos

El motor del algoritmo de enfriamiento simulado ha sido desarrollado utilizando algunos resultados ya publicados como [HUA86], [SWA90], [SEC88] y [BET99b].

La temperatura inicial se determina usando el mismo método de [HUA86], para ello se realizan N_c movimientos aleatorios, donde N_c es el número de celdas del circuito y se toma como temperatura inicial la desviación típica de los movimientos generados multiplicada por 20, asegurando así que al comienzo del proceso de enfriamiento simulado, prácticamente todos los movimientos son aceptados.

Para determinar el número de movimientos por temperatura, también llamado *condición de equilibrio*, se han usado los resultados obtenidos en [SWA90], donde el número máximo de movimientos por temperatura se calcula como:

$$Mov_{max} = K(N_c)^{4/3}$$

Donde K es una constante, cuyo valor determinará la calidad de la solución obtenida y el tiempo necesario para alcanzarla. Al igual que en [BET99b], para los experimentos que se presentan a continuación, se ha tomado $K = 10$.

En [LAM88] y [SWA90] se muestra que para una buena optimización usando la técnica de enfriamiento simulado, el porcentaje de movimientos aceptados frente al número total de movimientos generados (α) debe de ser de un 44%. Para lograr este objetivo, se ha adoptado la solución usada en [BET99b], donde el descenso de la temperatura en cada

iteración depende del valor de α:

$$T_k = \gamma(\alpha) \cdot T_{k-1}$$

Los valores de γ dependiendo de α pueden verse en la tabla 3.1.

α	γ
$\alpha > 0.96$	0.5
$0.8 < \alpha \leq 0.96$	0.9
$0.15 < \alpha \leq 0.8$	0.95
$\alpha \leq 0.15$	0.8

Tabla 3.1: Parámetro γ en función de α

Los resultados que se muestran a continuación han sido obtenidos usando el banco de circuitos de prueba desarrollado por el Centro Microelectrónico de Carolina del Norte [MCNC93], utilizado en el capítulo anterior. Para comparar los resultados obtenidos se han realizado los mismos experimentos con el más reconocido programa de colocación y rutado existente: VPR (Versatil Place and Route) [BET97a]. La arquitectura de la FPGA utilizada consiste en bloques lógicos complejos como el usado en el capítulo anterior, formado por 4 4-LUTs y 4 flipflops. Para probar el algoritmo en cualquier situación real, se han realizado experimentos sobre arquitecturas de dimensión mínima, dimensión fija y segmentos de varias longitudes.

En primer lugar, se ha querido demostrar la eficiencia del algoritmo realizando las pruebas publicadas en [BET97a], es decir, FPGA cuadrada de dimensiones mínimas (según el tamaño del circuito a probar) y segmentos de longitud unidad, pruebas generalmente aceptadas por todos los autores de artículos referentes a algoritmos de colocación de FPGAs. Las características de los circuitos utilizados en las pruebas se detallan

en la tabla 3.2 ordenados por el número de bloques lógicos. Para medir la optimalidad de la soluciones obtenidas, de nuevo se ha utilizado el programa VPR, el cual a partir de un circuito ya colocado determina el número mínimo de segmentos necesarios para conseguir rutar el circuito, este resultado puede utilizarse para medir la rutabilidad de la solución obtenida, cuanto mayor sea el número de segmentos necesarios para rutar un circuito, peor es su rutabilidad. Los resultados obtenidos para los dos algoritmos pueden verse en la tabla 3.3. Las diferencias entre los dos algoritmos para este tipo de arquitectura es mínima, la rutabilidad empeora en un $0,84\%$.

Circuito	Nº BL	Nº PADs	Nº Redes
MULT32A	18	34	101
TTT2	19	45	90
CLIP	32	14	113
MULT16B	34	18	106
APEX2	40	41	157
ALU	52	31	186
SAND	53	20	182
MISEX3C	58	28	195
PDC	60	56	211
PLANET	62	26	207
S1423	87	22	226

Tabla 3.2: Característica de los circuitos de prueba

Pero estas pruebas tienen un defecto, y es que no toman en cuenta la existencia de segmentos largos, como ocurre en todas las FPGAs comerciales, y tampoco indican el comportamiento de los algoritmos en FPGAs de dimensiones fijas, donde en muchas ocasiones el circuito es más pequeño que la FPGA y por lo tanto pueden aprovecharse recursos de rutado adyacentes para mejorar la rutabilidad. Las pruebas siguientes tratan de caracterizar el algoritmo tomando en cuenta los dos puntos anteriores. En el caso de los segmentos, para mayor generali-

Circuito	VPR	VGC
MULT32A	6	7
TTT2	7	7
CLIP	12	10
MULT16B	6	6
APEX2	11	11
ALU	12	13
SAND	14	13
MISEX3C	13	14
PDC	15	15
PLANET	14	15
S1423	9	9
Total	119	120

Tabla 3.3: Rutabilidad (FPGA cuadrada, dimensiones mínimas y segmentos de longitud 1)

dad, se han realizado experimentos para longitudes de segmentos entre 2 y 4 bloques lógicos (por cuántos bloques lógicos pasa el segmento en cuestión). Esto es, se han colocado y rutado los ejemplos anteriores en arquitecturas de rutado con segmentos de longitud 2, 3 y 4. En el caso del tamaño fijo, se han elegido las dimensiones de FIPSOC de 12×8 bloques lógicos. Los resultados obtenidos para segmentos de longitud 2 se muestran en la tabla 3.4, para segmentos de longitud 3 en la tabla 3.5 y los de longitud 4 en la tabla 3.6. Puede observarse un incremento en la rutabilidad cercano al 17% en el caso de segmentos de longitud 3, lo que confirma la optimalidad del algoritmo para arquitecturas *reales*.

Queda por ver el comportamiento del algoritmo en cuanto a retrasos y a tiempo de ejecución. En este paso del proceso de colocación y rutado, no existe una forma clara de determinar los retrasos de las conexiones, sin embargo, y sin entrar en arquitecturas de rutado específicas, puede medirse el número de segmentos que usa una red para conectarse; en principio, el retraso dependerá del número de segmentos que usa la red,

Circuito	VPR	VGC
MULT32A	7	5
TTT2	8	6
CLIP	11	8
MULT16B	6	5
APEX2	12	10
ALU	14	11
SAND	14	13
MISEX3C	14	13
PDC	16	14
PLANET	14	14
S1423	11	11
Total	127	110

Tabla 3.4: Rutabilidad (FPGA 12×8, segmentos de longitud 2)

Circuito	VPR	VGC
MULT32A	8	5
TTT2	9	6
CLIP	13	9
MULT16B	8	5
APEX2	13	10
ALU	15	12
SAND	16	14
MISEX3C	15	14
PDC	16	15
PLANET	17	15
S1423	12	13
Total	142	118

Tabla 3.5: Rutabilidad (FPGA 12×8, segmentos de longitud 3)

Circuito	VPR	VGC
MULT32A	9	6
TTT2	9	7
CLIP	13	9
MULT16B	9	6
APEX2	14	11
ALU	16	13
SAND	16	15
MISEX3C	17	15
PDC	17	16
PLANET	17	15
S1423	13	14
Total	150	127

Tabla 3.6: Rutabilidad (FPGA 12×8, segmentos de longitud 4)

cuanto mayor sea éste, mayor será el número de interruptores programables utilizados. Teniendo en cuenta que estos elementos presentan resistencias y capacidades parásitas muy superiores al metal usado en las conexiones, dicho número puede darnos una idea de cómo será el retraso. En la tabla 3.7 pueden verse el tiempo de ejecución y el número medio de segmentos usados por las redes para la típica arquitectura de rutado: segmentos de longitud unidad y FPGA cuadrada de dimensiones mínimas. Este tiempo ha sido medido en un ordenador personal con procesador AMD-K6 a 500MHz y con 128 MB de memoria RAM. Como era de esperar, el tiempo de ejecución para el nuevo algoritmo supera con creces el tiempo del VPR, debido a que sólo calcula el semiperímetro de las redes mientras que el VGC debe estimar la congestión de los canales de rutado. En cuanto a retrasos, el número medio de segmentos es muy parecido por lo que es de esperar un buen comportamiento en este caso.

	Tiempo(s)		Num. Segmentos	
Circuito	VPR	VGC	VPR	VGC
MULT32A	2	17	1,88	2,00
TTT2	3	22	2,40	2,60
CLIP	3	25	3,12	2,91
MULT16B	2	21	2,12	2,20
APEX2	5	47	2,89	2,84
ALU	6	76	3,05	3,24
SAND	4	60	3,37	3,38
MISEX3C	7	74	3,31	3,46
PDC	9	108	3,43	3,50
PLANET	5	97	3,32	3,34
S1423	6	120	2,61	2,91

Tabla 3.7: Tiempo de ejecución y retrasos

3.4 Conclusiones

Se ha propuesto un nuevo algoritmo de colocación que trata de maximizar la rutabilidad de los circuitos basándose en una estimación simple del camino seguido por las conexiones dentro de la FPGA. Las ventajas ofrecidas por este algoritmo son las siguientes:

1. Permite definir asimetrías dentro de la FPGA de una forma sencilla, lo cual es importante en los SOCs.

2. En arquitecturas simétricas y próximas a las utilizadas en FPGAs comerciales, ofrece un buen comportamiento en cuanto a rutabilidad, hallando soluciones hasta un 17% más rutables.

3. El retraso de las conexiones es similar al conseguido por un algoritmo basado en semiperímetro.

Como desventaja, cabe señalar el aumento del tiempo de ejecución aunque los resultados obtenidos demuestran que los tiempos son aceptables.

Capítulo 4

Rutado

Dentro de las herramientas de síntesis automática, el rutado es la etapa previa a la generación del flujo de bits que se transmitirá a la FPGA para su programación. Después de la colocación de los bloques lógicos, debe elegirse el estado de cada uno de los interruptores programables para conformar las distintas conexiones entre dichos bloques. Podría decirse que esta etapa opera con el nivel de abstracción más bajo del dispositivo, es decir, con la programación de los transistores que forman los interruptores.

Debido a la gran variedad de trabajo realizado a lo largo del desarrollo de esta tesis, el autor ha querido englobar en este capítulo, no sólo la algorítmica asociada al problema de rutado, si no también los estudios relacionados con la arquitectura interna del dispositivo programable.

La componente principal del coste de una FPGA es el área de silicio que ocupa. Por ello, un objetivo de su diseño es la reducción del área. En este capítulo se presenta un método para la reducción del área de silicio mediante el uso de decodificadores en los pines de salida de los bloques lógicos. En una primera parte, se desarrollará un modelo de

área que posteriormente servirá para el estudio en cuestión. Finalmente se estudiarán las consecuencias del uso de dichos decodificadores en las herramientas de rutado y se presentará una solución a los problemas encontrados.

4.1 Introducción

En 1993, Patrick Lysaght y Jon Dunlop [LYS93] definían por primera vez algunos conceptos aplicables a dispositivos programables. Para estos autores las FPGAs podían clasificarse según su configurabilidad como:

- Programables: todas las FPGAs permiten ser programadas, aunque sólo sea una vez.
- Reconfigurables: Aquellas FPGAs que tras su funcionamiento pueden volver a ser programadas.
- Parcialmente Reconfigurables: Permiten programar de forma selectiva alguna parte de la FPGA mientras que el resto guarda la configuración anterior, realizándose dicha operación cuando el dispositivo está inactivo.
- Dinámicamente Reconfigurables: Igual que el anterior pero la operación puede realizarse mientras el dispositivo está activo.

En aquel tiempo no existía ningún dispositivo parcialmente o dinámicamente reconfigurable.

En los últimos años, han aparecido nuevas familias de FPGAs como la XC6200 [XIL97] y FIPSOC [FAU97]. Estas FPGAs *dinámicamente reconfigurables* permiten seleccionar parte de sus elementos de memoria de configuración y cambiarlos de estado sin afectar al funcionamiento

del resto del dispositivo. FIPSOC es además el primer dispositivo comercial *multicontexto* (aunque ya existían algunos prototipos como la DPGA [TAU95]). Básicamente consiste en que la memoria de configuración del dispositivo está repetida un cierto número de veces, tantas como contextos, y una señal se encarga de seleccionar qué memoria de configuración actúa sobre el dispositivo en un momento determinado. En cada instante existe un único contexto activo que configura el dispositivo completo; un cambio de contexto permite cambiar parcial o totalmente la funcionalidad del dispositivo. En el caso de su aplicación a FPGAs, se obtendría una multiplexación en el tiempo de la funcionalidad de dicha FPGA.

El problema de los dispositivos multicontexto, como ya reflejaba André DeHon en su tesis doctoral [DEH96], es la cantidad de área de silicio consumida por la memoria de configuración, que en el caso de una FPGA normal, podía llegar hasta un 80% del área total de silicio. En el caso de una FPGA multicontexto, esta situación se agrava ya que la memoria de configuración es repetida tantas veces como contextos tenga.

Para resolver dicho problema, André DeHon apunta una solución que podría reducir el área de silicio: la codificación de la memoria de configuración. Esta técnica ya ha sido utilizada por la mayoría de los fabricantes de FPGAs para la memoria de configuración de los pines de entrada de los bloques lógicos. Cada pin de entrada debe ser conectado a un segmento del canal de rutado; como normalmente existe sólo una salida que se conecta a varias entradas, a cada pin de entrada le llegará una sola señal. Unos interruptores programables permiten seleccionar a qué segmento se conectan los pines de entrada. Por lo dicho anteriormente, sólo uno de estos interruptores estará activo al mismo tiempo.

En la figura 4.1 puede verse un ejemplo en el que un pin de entrada puede conectarse a 8 segmentos diferentes de un mismo canal de rutado. Si cada interruptor programable está controlado individualmente, se necesitarán 8 celdas de memoria para configurar la entrada. Como este pin de entrada sólo se va a conectar a un segmento, sólo uno de estos interruptores estará activo en un instante. Usando un decodificador (ver figura 4.2), la cantidad de memoria necesaria se reduce de 8 a 3 bits.

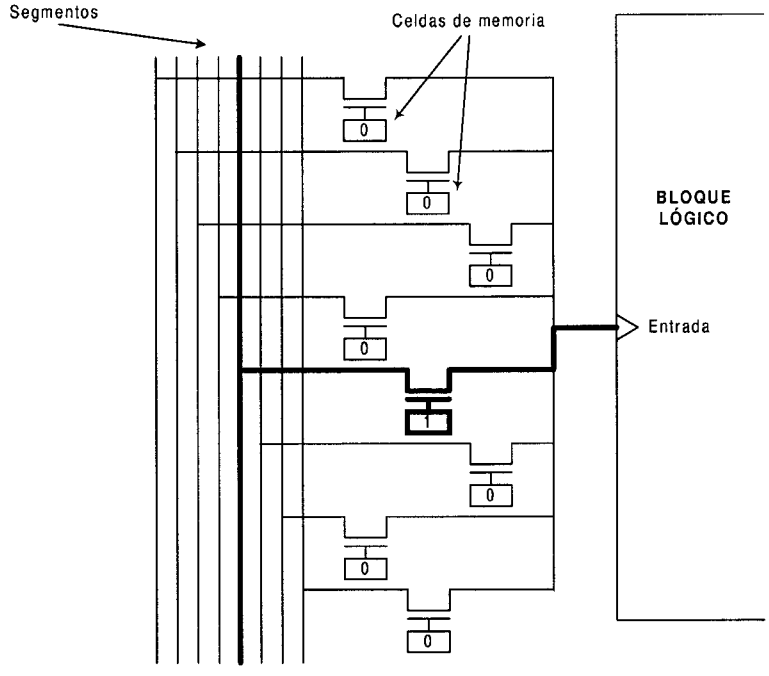


Figura 4.1: Entrada sin decodificador (8 celdas de memoria)

En este capítulo se pretende aplicar la misma técnica a los pines de salida [BAE99a], donde se plantea un problema adicional: los pines de salida sí pueden estar conectados a varios segmentos a la vez, ya que la señal puede ir a varias entradas. Al utilizar decodificadores en los pines de salida, se restringe la conexión de estos a un único segmento al mismo tiempo, por lo que se plantea una serie de cuestiones:

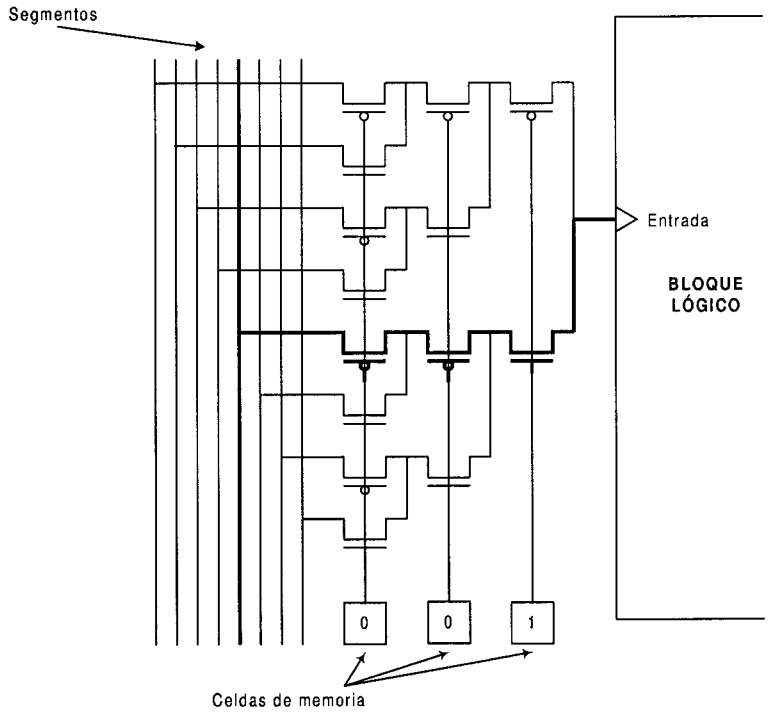


Figura 4.2: Entrada con decodificador (3 celdas de memoria)

1. Rutabilidad: puede reducirse la rutabilidad de la arquitectura de la FPGA debido a la restricción anteriormente dicha.

2. Velocidad: dependiendo de la solución empleada, el retraso de las señales puede aumentar. ¿En qué medida es asumible?

3. Herramientas de rutado: No existe ninguna herramienta de rutado que permita el uso de decodificadores a la salida de los bloques lógicos.

4. Área: ¿Cuál es la reducción en área conseguida al emplear dicha técnica?

En las próximas secciones, se tratará de dar solución a cada uno de estos problemas comenzando por la definición de una arquitectura genérica que permita el estudio de los problemas anteriores. Seguidamente se

desarrollará un modelo de área para evaluar la reducción en área del dispositivo al emplear decodificadores en los pines de salida. Posteriormente se presentará la herramienta de rutado utilizada junto con el procedimiento empleado para el estudio de retrasos, y por último se presentarán los resultados obtenidos.

4.2 Arquitectura

Utilizaremos una arquitectura de FPGA muy genérica, por lo que los resultados presentados en este capítulo pueden ser extrapolados fácilmente a la mayoría de las FPGAs existentes en el mercado. Consiste en una matriz cuadrada de bloques lógicos con canales de rutado horizontales entre las filas y canales de rutado verticales entre las columnas. Estos canales de rutado son idénticos y contienen el mismo número de líneas de segmentos: W. Para mayor simplicidad, cada línea está compuesta por un cierto número (dependiendo del tamaño de la matriz) de segmentos de longitud unidad, conectados a otros segmentos a través de bloques S. En la figura 4.3 pueden verse resumidas las propiedades anteriores. Este modelo de arquitectura es el utilizado en la gran mayoría de trabajos publicados sobre FPGAs.

Los bloques lógicos utilizados están formados por un determinado número de elementos lógicos básicos (ELB), descritos ya en capítulos anteriores, que serán recordados aquí para mayor facilidad de lectura. En la figura 4.4 puede verse un ELB y el pinout de un bloque lógico formado por un solo ELB. Cada ELB está compuesto por una LUT de 4 entradas y un biestable. Estudios anteriores han demostrado que este tipo de bloque lógico conduce a la mayor eficiencia en área [ROS90], y la mayoría de las FPGAs comerciales están basadas en este tipo de bloque lógico [XIL99] [ALT99]. La salida de la 4-LUT, es accesible a través del pin O0

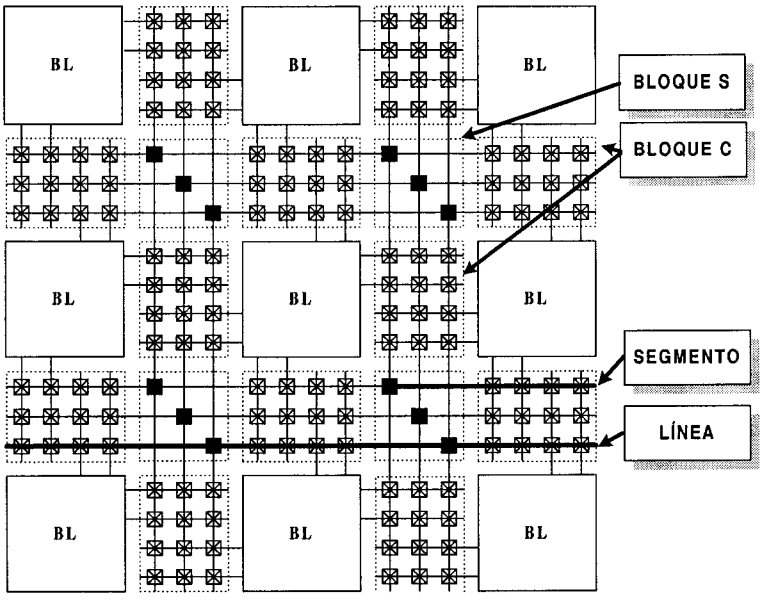


Figura 4.3: Arquitectura de la FPGA

del bloque lógico mientras que la salida del biestable puede ser leída a través del pin O1.

Para mayor simplicidad, el parámetro F_c de los bloques C, es decir el número de segmentos a los cuales puede conectarse un pin de un bloque lógico, es igual a W (el número de líneas de segmentos por canal). El parámetro F_s de los bloques S es igual a 3. Como ya se ha dicho anteriormente, estos valores son muy cercanos a los valores teóricos ideales hallados en [ROS91], donde F_c tiene un valor entre $0.7 \cdot W$ y $0.9 \cdot W$, y $F_s = 3$. En la figura 4.5 puede verse con mayor detalle la estructura de un bloque S con $F_s = 3$.

4.3 Modelo de área

En esta sección se desarrollará un modelo de área para estudiar y cuantificar el ahorro de silicio al usar decodificadores en los pines de salida

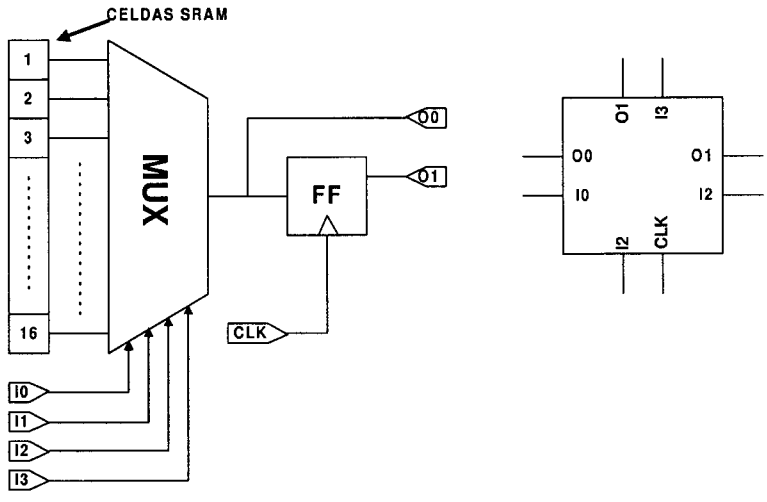


Figura 4.4: Estructura interna de un bloque lógico

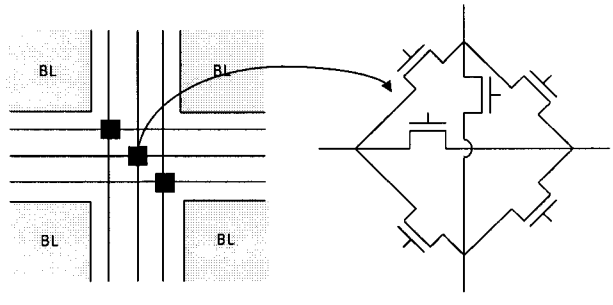


Figura 4.5: Estructura interna de un bloque S

de los bloques lógicos [BAE99b]. El área de un bloque lógico puede ser dividida en tres grandes categorías:

1. Área Fija: que tiene en cuenta el área del multiplexor de la LUT de 4 entradas y el biestable.
2. Área de datos: que corresponde al área de las 16 celdas de memoria de la LUT.
3. Área de interconexión: el área ocupada por los bloques S y C de un bloque lógico.

Asumiendo una tecnología de 3 metales, el área fija puede estimarse en $13K\lambda^2$ según [BRO92b] o en $20K\lambda^2$ según [DEH96] donde λ es la constante de la tecnología. El área de datos depende del tipo de celda de memoria utilizada; si se utilizan celdas RAM estáticas (SRAM), un valor típico del área de estas celdas es de $1200\lambda^2$ [DEH96]. Si C es el número de contextos de la FPGA, el área total de datos del bloque lógico es de:

$$A_{datos} = 16 \cdot C \cdot A_{mem}$$

siendo A_{mem} el área de una celda de memoria.

El área de interconexión se divide en el área de los bloques C y en el área de los bloques S. En el caso de los bloques C, las conexiones de los pines pueden realizarse de dos formas distintas dependiendo de si usan o no decodificadores. Si no utilizan decodificadores, debe realizarse una conexión entre dicho pin y F_c segmentos mediante F_c interruptores programables usando F_c celdas de memoria. Estos interruptores son relativamente pequeños (transistores de paso o puertas de transmisión) frente a las celdas de memoria, por lo que puede asumirse que, en este

caso, el área de las celdas de memoria es predominante (el rutado y las conexiones pueden realizarse encima de la memoria). Por lo tanto el área de conexión entre un pin de un bloque lógico y F_c segmentos para una FPGA con C contextos es:

$$A_{conx} = C \cdot F_c \cdot A_{mem}$$

Si el pin usa decodificadores, en el peor de los casos, no se podrá adoptar la solución anterior y el área será la suma del área de la memoria y del rutado. El área de la memoria en este caso puede estimarse fácilmente en:

$$C \cdot log_2(F_c) \cdot A_{mem}$$

Si se utilizan las salidas negadas de las celdas de memoria, se pueden utilizar sólo transistores NMOS, con el correspondiente ahorro de los pozos tipo N para los PMOS (ver figura 4.6). Si WP es el ancho mínimo de una pista de metal (con un valor típico de 6–8λ, dependiendo del nivel de la capa de metal), cada conexión de un pin a un segmento puede estimarse en:

$$2 \cdot WP \cdot WD \cdot log_2(F_c)$$

donde WD es el ancho del decodificador para una conexión (ver figura 4.6). Realizando un diseño físico de este decodificador, puede conseguirse fácilmente una WD menor de $4 \cdot WP$. Por lo tanto, el área de las F_c conexiones de un pin en un bloque C puede estimarse como:

$$8 \cdot F_c \cdot WP^2 \cdot log_2(F_c)$$

El área de conexión entre un pin de un bloque lógico y F_c segmentos usando un decodificador para una FPGA con C contextos es, por lo tanto,

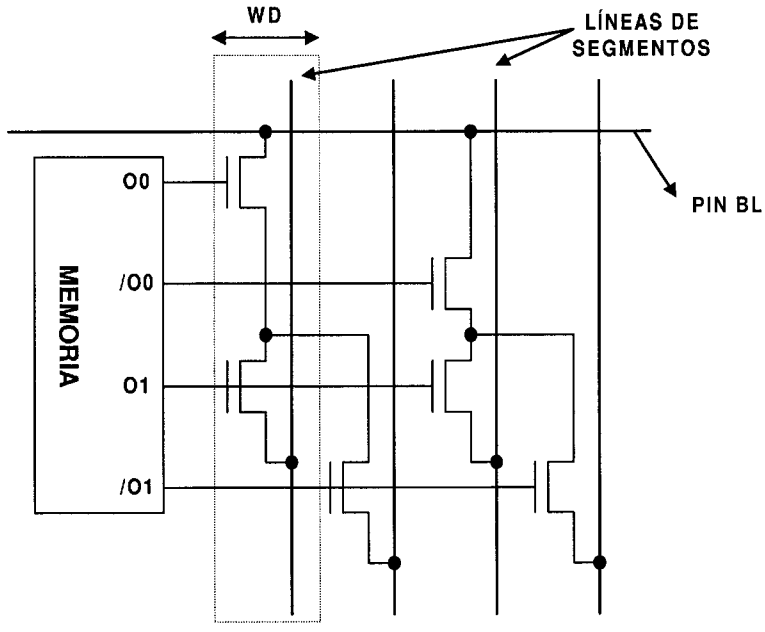


Figura 4.6: Decodificador para un pin de salida

la suma del área de la memoria más el área del decodificador:

$$A_{conx} = C \cdot log_2(F_c) \cdot A_{mem} + 8 \cdot F_c \cdot WP^2 \cdot log_2(F_c)$$

Para los bloques S, si no se usa ningún tipo de codificación para la memoria de configuración, el área predominante será la de la memoria por lo que el área total puede calcularse mediante:

$$A_{bs} = 2 \cdot F_s \cdot W \cdot C \cdot A_{mem}$$

El uso de decodificadores o multiplexores para las entradas de los bloques lógicos es muy común en FPGAs comerciales, debido a que, como se ha dicho, sólo se conecta un segmento a dichos pines. Por esta razón, de ahora en adelante, se supondrá el uso de dichos decodificadores para los pines de entrada de los bloques lógicos, ya que no tiene sentido tratar de ahorrar área utilizando decodificadores en los pines de salida si no se ha hecho con los de entrada.

Si NE es el número de entradas al bloque lógico, y NS el número de salidas, el área ocupada por un bloque lógico sin usar decodificadores en los pines de salida puede modelarse mediante:

$$\begin{aligned} A_{total} &= A_{fija} + 16 \cdot C \cdot A_{mem} + \\ &\quad NS \cdot F_c \cdot C \cdot A_{mem} + \\ &\quad NE \cdot log_2(F_c) \cdot (C \cdot A_{mem} + 8 \cdot W \cdot WP^2) + \\ &\quad 2 \cdot F_s \cdot W \cdot C \cdot A_{mem} \end{aligned}$$

Para un bloque lógico con decodificadores en las salidas:

$$\begin{aligned} A_{total} &= A_{fija} + 16 \cdot C \cdot A_{mem} + \\ &\quad (NE + NO) \cdot log_2(F_c) \cdot (C \cdot A_{mem} + 8 \cdot W \cdot WP^2) + \\ &\quad 2 \cdot F_s \cdot W \cdot C \cdot A_{mem} \end{aligned}$$

Mediante estas ecuaciones se ha realizado un estudio del ahorro en área producido por el uso de los decodificadores, el cual se detalla a continuación.

Suponiendo $A_{mem} = 1200\lambda^2$, $WP = 8\lambda$ y $A_{fija} = 20K\lambda^2$ (el peor caso), la figura 4.7 presenta el porcentaje de reducción de área obtenido usando el modelo de área anteriormente descrito para diferentes valores de F_c y C (número de contextos). Puede observarse que el máximo ahorro en área para esta arquitectura es de un 9% para $C = 6$ y $F_c = 63$. Analizando cuidadosamente los diferentes términos que aparecen en el modelo de área, se observa que en este caso, los bloques S dominan el área de interconexión.

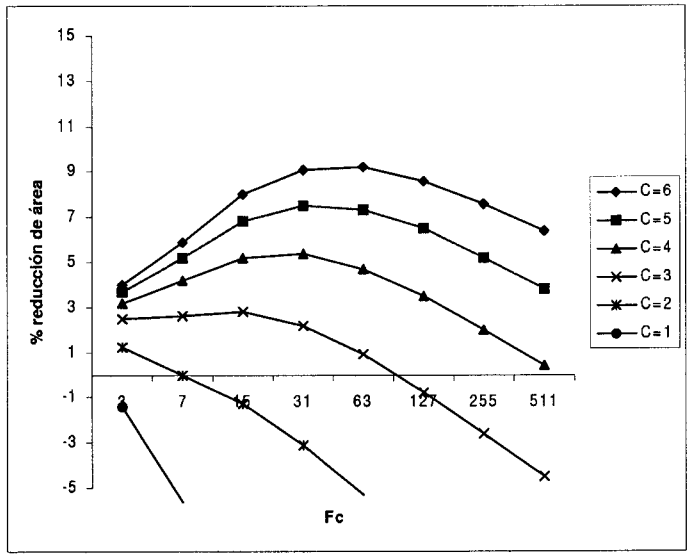


Figura 4.7: Reducción de área para un bloque lógico formado por 1 ELB

Si se considera una estructura de bloque lógico más compleja, formada por varios elementos lógicos básicos (ELB), los resultados cambian significativamente. Si N es el número de ELBs que contiene el bloque lógico, para $N = 2$ (figura 4.8), lo cual corresponde a una arquitectura similar a la empleada en la serie 3000 de Xilinx, la máxima reducción en área es de un 17,2%. Para $N = 3$ (figura 4.9), bloque lógico similar al empleado en la serie 4000 de Xilinx, la reducción se sitúa en un 19,6%. Por último, para $N = 4$ (figura 4.10), el tipo de bloque lógico utilizado en FIPSOC, se alcanza un ahorro en área de un 21%.

Claramente, para FPGAs de grano grueso y un número de contextos igual o superior a 3, el ahorro en área producido por el uso de la técnica anterior es considerable. Por este motivo, **si lo que se persigue es el ahorro en área**, ninguna de las FPGAs comerciales citadas anteriormente, se beneficiarían del uso de decodificadores en los pines de salida de los bloques lógicos. De estas, FIPSOC es el único dispositivo

multicontexto, pero sólo posee dos. Queda por ver el efecto de los decodificadores en la velocidad y la rutabilidad del dispositivo, lo cual será tratado en las secciones 4.4 y 4.5 de esta tesis.

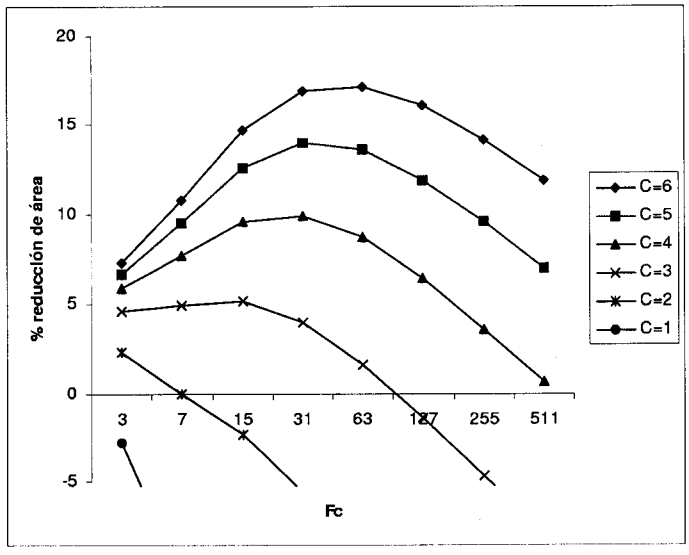


Figura 4.8: Reducción de área para un bloque lógico formado por 2 ELBs

4.4 Herramientas de rutado

El uso de decodificadores en las conexiones de los pines de salida de los bloques lógicos con los segmentos de los canales de rutado claramente reduce la rutabilidad. En general, estos pines deben acceder a varios pines de entrada de otros bloques lógicos, por lo que restringir las conexiones de estos pines a un único segmento disminuye la probabilidad de encontrar un camino entre ese pin y los restantes. Para medir ese decremento en rutabilidad se va a desarrollar una nueva herramienta de rutado que posibilite definir decodificadores en los pines de salida.

Hasta la fecha de redacción de esta tesis, sólo existe una herramienta de rutado que permite el uso de estos decodificadores: RAISE [BAE98]. Sin embargo, este algoritmo presenta un grave problema de cara al usuario

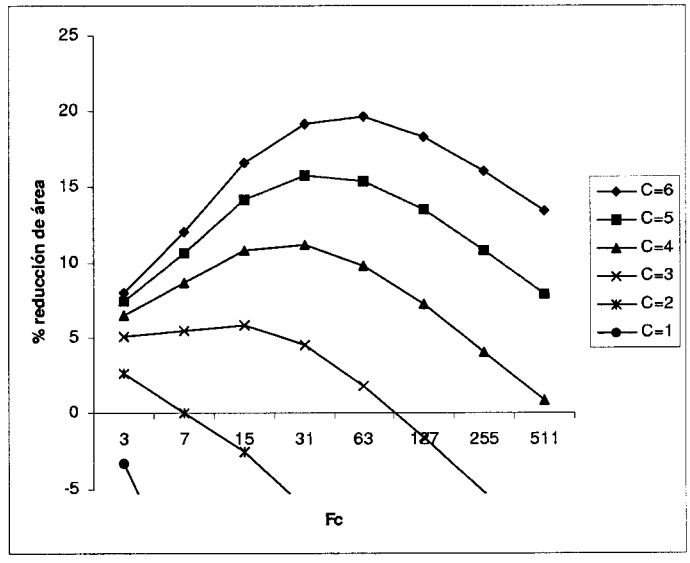


Figura 4.9: Reducción de área para un bloque lógico formado por 3 ELBs

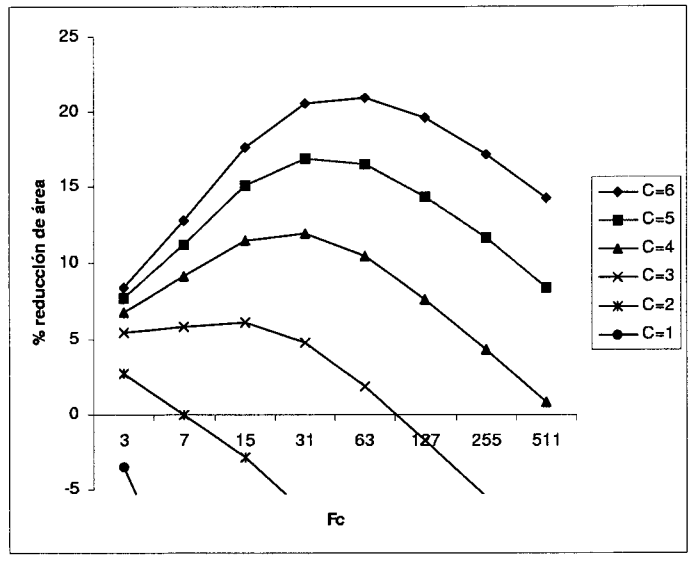


Figura 4.10: Reducción de área para un bloque lógico formado por 4 ELBs

final: el elevado tiempo de ejecución, del todo inaceptable para una herramienta comercial. Por este motivo, se ha creido necesario desarrollar un nuevo algoritmo, con un tiempo de ejecución más bajo. En realidad, para simplificar la tarea (en principio), se ha optado por adaptar alguna de las herramientas existentes, como VPR [BET97a] o SEGA [LEM93], para la resolución del problema.

4.4.1 Adaptación de VPR

Los resultados publicados por *Vaughn Betz*, autor del VPR, demuestran la optimalidad de su herramienta. Pero la base del algoritmo de rutado imposibilita su uso para los decodificadores. El programa utiliza una variación del conocido algoritmo del *laberinto* o *maze* llamado *PathFinder* [EBE95], básicamente consiste en lo siguiente:

1. Se le asigna un coste no nulo a todos los elementos de rutado (segmentos).
2. Se selecciona de forma aleatoria una red y se busca el camino de coste mínimo usando el algoritmo del laberinto. Esto se repite hasta que no quede ninguna red por rutar.
3. Seguramente, existirán algunos segmentos que hayan sido utilizados por varias redes lo que conlleva a una solución errónea. A estos recursos compartidos por redes distintas se les incrementa su coste y se retorna al punto segundo.
4. Si no existen recursos compartidos por varias redes, la solución es legal, y termina el algoritmo.

El problema que plantea este algoritmo se encuentra en el punto 2. Una vez que se ha seleccionado aleatoriamente la red, hay que proceder a su

rutado. Si la red es *monopunto*, es decir, posee una fuente y un único sumidero, la aplicación del algoritmo del laberinto es trivial. Pero si la red es *multipunto*, una fuente y varios sumideros, esto se complica.

Para comprender mejor el por qué de esa complicación, se ha representado gráficamente la expansión del algoritmo del laberinto para una red multipunto compuesta de una fuente $S1$ y dos sumideros $E1$ y $E2$. En la figura 4.11 puede observarse la situación de partida donde se ha simplificado la arquitectura de rutado de la FPGA y sólo se representan los elementos de interés. Puede verse que aunque en la sección sobre la arquitectura de rutado se dijo que se consideraría una arquitectura con una $F_c = W$, en este caso $F_c = 2$ y $W = 3$, esto se debe a que el problema que se comentará a continuación sólo aparece cuando F_c es distinto de W, algo que puede ocurrir en la realidad y que, por lo tanto, debe ser tomado en cuenta, aunque sólo sea en esta sección, ya que la herramienta que se quiere desarrollar debe de ser aplicable a cualquier arquitectura.

En la figura 4.12 se puede observar el siguiente paso, donde ya se han seleccionado dos segmentos más. En la figura 4.13 aparece la siguiente expansión. En la próxima iteración, se encontrarán dos segmentos que podrán conectarse a $E1$, tendremos que elegir uno de ellos. Supongamos que se elige el de la izquierda (por ahora no existe ninguna razón que nos indique lo contrario) y que llegamos a la situación de la figura 4.14, puede observarse que si se elige el camino señalado para conectar $S1$ con $E1$, ya no existe ninguna posibilidad de conectar $S1$ con $E2$ si $S1$ utiliza un decodificador para conectarse al segmento. Si se hubiera rutado la conexión $S1$-$E2$ primero, no existiría dicho problema, pero ¿cómo se pueden ordenar *a priori* las conexiones para encontrar una solución posible? No se ha encontrado una solución a esta pregunta. Se ha tra-

tado de realizar un ordenamiento aleatorio, pero el algoritmo se volvía inestable, es decir, no existía una convergencia hacia una solución en un tiempo finito.

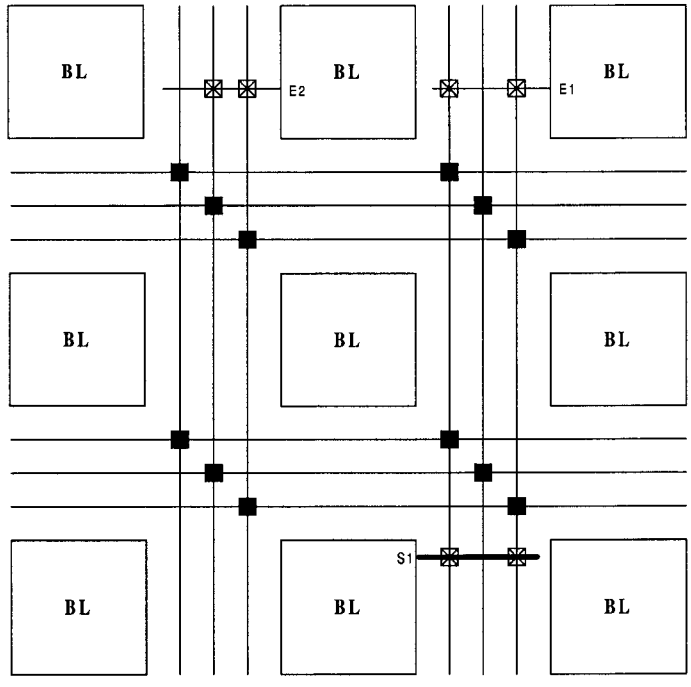


Figura 4.11: Expansión 1

Otra posibilidad es tratar de rutar todas las conexiones de una sola vez, es decir, en el último paso anterior, se seguiría la expansión hasta llegar hasta $E2$. Esta posibilidad presenta dos problemas, el primero y el más evidente es que cada vez que se añade un segmento a la expansión se va ocupando más memoria. Si a esto se le añade que en FPGAs reales, los pines a conectar pueden estar en zonas muy remotas las unas de las otras, la memoria y el tiempo empleado con esta técnica puede llegar a ser muy elevado (suponiendo una $F_s = 3$, el número de segmentos pertenecientes a la expansión n sigue un crecimiento igual 3^n). El otro problema, no tan evidente, tiene que ver con la topología del bloque S. El

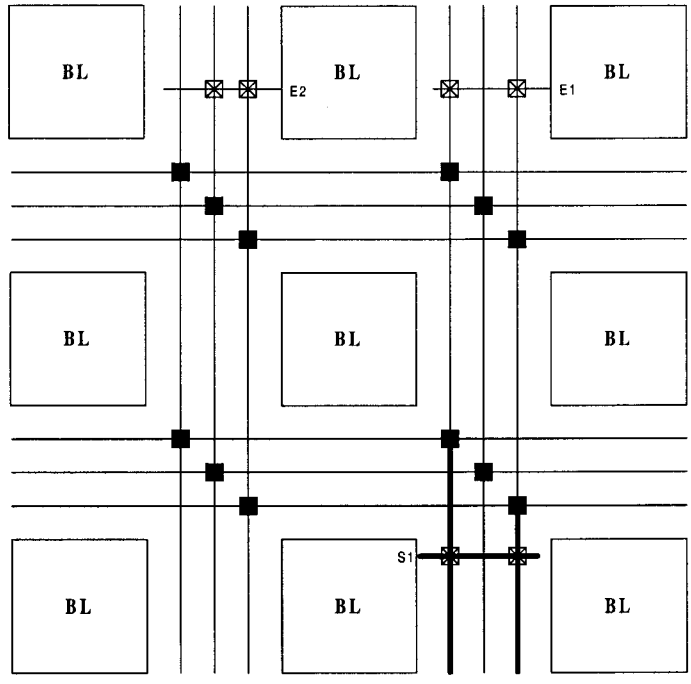


Figura 4.12: Expansión 2

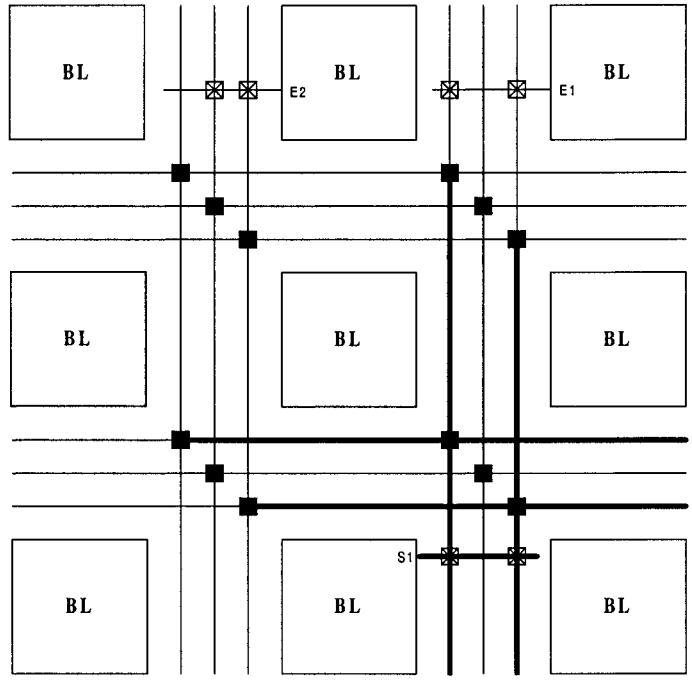


Figura 4.13: Expansión 3

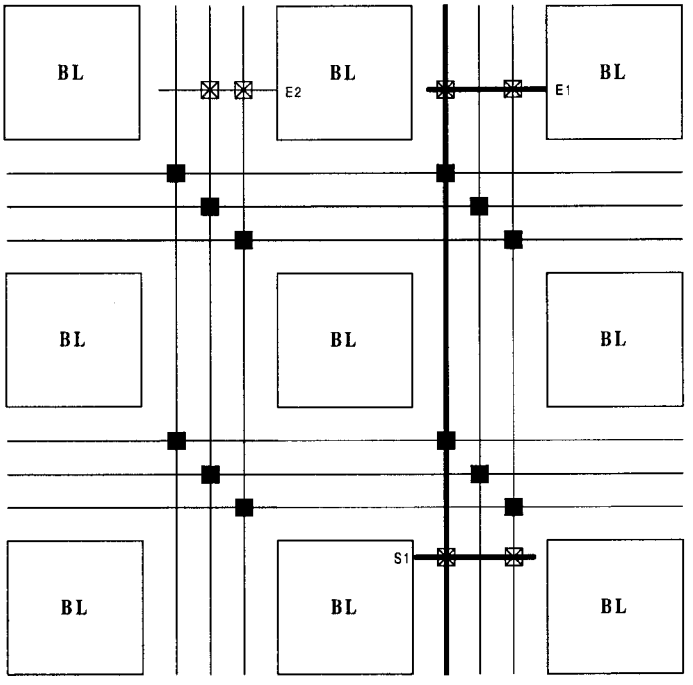


Figura 4.14: Expansión 4

bloque S hasta ahora empleado es tal que una red que use un segmento perteneciente a una determinada línea de segmentos, por ejemplo un segmento del canal de rutado vertical y de la línea de segmentos de la izquierda en la figura 4.14, sólo usará segmentos verticales pertenecientes a dicha línea o la equivalente en otro canal de rutado vertical o la línea superior en los canales de rutado horizontales. Esto quiere decir que si en la expansión llegamos a un segmento, podemos marcarlo para no volver a incluirlo en una expansión posterior y no formar bucles, ya que como sólo existe una forma de llegar a él (a partir de un solo segmento inicial) y ya hemos encontrado el camino más corto por el propio funcionamiento del algoritmo, no deberemos pasar de nuevo por él. Sin embargo, si la topología del bloque S es distinta a la anterior, lo cual puede ocurrir [FAU97], un segmento de una línea de segmentos determinada puede ser alcanzado por segmentos pertenecientes a otras

líneas, por lo que no se puede marcar el segmento tan fácilmente la primera vez que se llega a él. Se necesita un proceso mucho más complejo que aumenta tremendamente la memoria consumida y el tiempo de ejecución. En la figura 4.15 se representa este caso donde las líneas punteadas representan las posibles conexiones que pueden realizarse entre segmentos mediante interruptores programables. Puede observarse que a partir de los dos segmentos conectados a la salida $S1$ puede alcanzarse el segmento conectado a $E2$. Por estas razones se ha desestimado la adaptación de la herramienta VPR para el uso de decodificadores en los pines de salida.

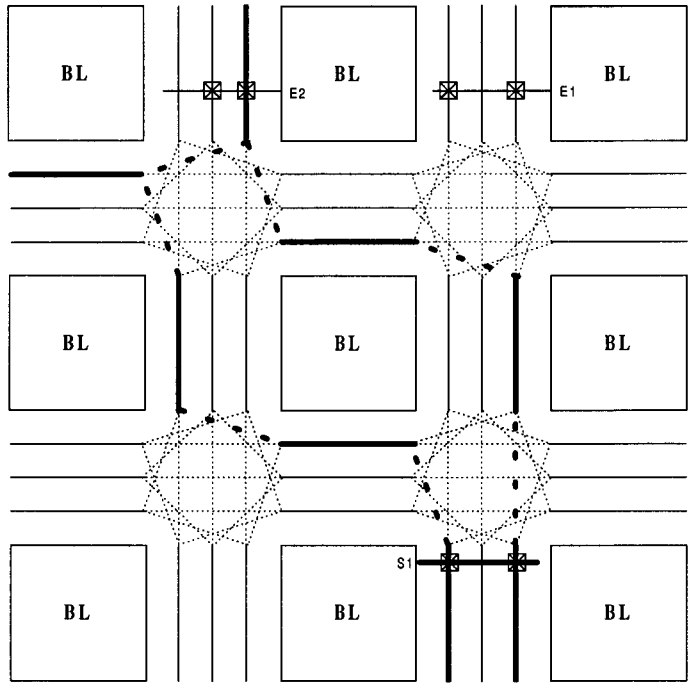


Figura 4.15: Otra topología para los bloques S

4.4.2 Adaptación de SEGA

La herramienta de rutado SEGA citada anteriormente, ofrece buenos resultados, aunque se diferencia de la anterior en que necesita de una

etapa previa: el rutado global, que se encarga de seleccionar los canales de rutado por donde las distintas redes tendrán que pasar. Tras este proceso, SEGA trata de hallar los segmentos que van a posibilitar la conexión entre los distintos pines utilizando un algoritmo desarrollado en [BRO92c] y [BRO92a] denominado *CGE*.

Básicamente, el algoritmo *CGE* (*Coarse Graph Expansion*), halla para cada red monopunto *todos* los posibles caminos para la conexión de los dos terminales, donde *todos* se refiere a los caminos que utilizan los canales de rutado dados por el rutador global. Una vez realizado esto, cada red posee un conjunto de caminos posibles. A partir de estos conjuntos se crea una *lista de caminos* que contiene todos los caminos de todas las redes. Seguidamente se procede de la siguiente forma:

1. Asignar un *coste* (ver más adelante) a cada camino
2. Si la lista esta vacía ir al punto 10
3. Si existe alguna red *esencial* (ver más adelante) seleccionar el camino de esa red con menor *coste*
4. Si no, seleccionar de la lista el camino con menor *coste*
5. Marcar la red correspondiente al camino seleccionado como rutada
6. Borrar todos los caminos de dicha red salvo el seleccionado
7. Encontrar todos los caminos que pudieran tener conflictos con el camino seleccionado y borrarlos, si una red se queda sin caminos posibles el circuito no se puede rutar, ir al punto 10.
8. Actualizar *costes*
9. Volver al punto 2

10. Fin

El término *esencial*, viene a referirse a las redes cuyo conjunto de caminos posibles se reduce a uno, es decir, en el paso 7 algunos caminos son borrados como consecuencia de que estos son incompatibles con otros que ya han sido seleccionados, por lo que puede ocurrir que alguna de las redes se quede sin caminos o con uno sólo. Esas redes esenciales deben rutarse de inmediato, sin importar el valor de su coste ya que no existen otras alternativas.

Aunque el funcionamiento del algoritmo aquí presentado difiere sustancialmente del que aparece en [BRO92a], un estudio detallado del código fuente de SEGA ha demostrado que el proceso referido más arriba es el que ha sido implementado y no el que aparece en la tesis anteriormente citada. Como mayor diferencia cabe destacar que en SEGA no aparece ningún mecanismo de *podado* o *prunning* que reduzca la lista de caminos. El proceso de podado, descrito en la tesis de Brown, consiste en eliminar algunas de las soluciones encontradas para los distintos caminos, reduciendo así la memoria utilizada y acelerando la ejecución de la herramienta. Esta eliminación puede basarse en algunas reglas heurísticas como la maximización de la entropía de las soluciones que no se eliminan, es decir, tratar de que los caminos guardados para una red utilicen segmentos diferentes para así disminuir la probabilidad de que esa red se quede sin caminos posibles, o simplemente realizar una eliminación aleatoria. Teóricamente, si tras la operación de podado no se alcanza ninguna solución para el rutado del circuito, se volvería a iniciar el algoritmo relajando los parámetros de control de ese podado, hasta alcanzar una solución.

Queda por definir la función de coste, que será de vital importancia a

la hora de tratar los decodificadores. La función de coste utilizada en SEGA tiene dos objetivos:

1. El camino seleccionado debe de suponer un pequeño decremento en la rutabilidad de las conexiones restantes, es decir, debe impedir la selección de un camino que contenga segmentos con una gran demanda.

2. Permite identificar una red esencial para su rutado inmediato.

El coste de un camino se calcula mediante la suma de los costes de los segmentos que lo componen. Para determinar si un segmento tiene una demanda elevada se podría contar cuántas veces aparece dicho segmento en la lista de caminos, pero eso no representa de forma adecuada la demanda ya que aunque el segmento aparezca muchas veces en la lista de caminos, puede ser que existan otras muchas alternativas para el rutado de una conexión. Por esta razón el coste de un segmento se define de la siguiente manera:

$$C(s_i) = \sum_{j \neq i} \frac{1}{alt(s_j)}$$

donde s_i es el segmento de la conexión i al que se le quiere calcular el coste, j son las conexiones que poseen al segmento s en su conjunto de caminos y que además no pertenecen a la misma red, y $alt(s_j)$ es el números de alternativas que no usan s para la conexión j. Debido al sumatorio, cuanto más veces aparece s en las soluciones del resto de las redes, mayor será su coste, por lo que se tratará de no utilizarlo. En el caso especial de que un segmento sólo aparezca en el conjunto de caminos de la conexión i, el coste se definirá como nulo, mientras que cuando una conexión no posea alternativas a s, su coste será infinito. Para cada camino del conjunto de soluciones de una conexión, se calculará su coste

como la suma de los costes de los segmentos que contiene.

Para adaptar esta función de coste al caso de los decodificadores, se ha añadido un término a dicha función de coste. El coste de un camino debe incluir el hecho de que si éste es seleccionado, todas las demás conexiones pertenecientes a esa red deberán utilizar el mismo segmento de salida. Para ello se ha introducido el siguiente término:

$$C_D(s_i) = \sum_{j \neq i} \frac{1}{coinc(s_j)}$$

donde s_i es el primer segmento (el que se conecta al pin de salida) de la conexión i, j son todas las conexiones monopunto de la red multipunto a la que pertenece i, y $coinc(s_j)$ es el número de caminos, que la conexión j posee, que usan el segmento s. Veamos un ejemplo para comprender su funcionamiento. Supongamos una red multipunto formada por n conexiones monopunto y supongamos además que queremos calcular el coste de un camino de una de esas conexiones. Si el camino elegido comienza con un segmento que no aparece en las soluciones de las otras conexiones, el coste de ese camino debería ser muy alto ya que si lo seleccionamos, no existirán soluciones para las demás conexiones de la misma red. En ese caso, como el segmento en cuestión no aparece en ninguna conexión: $coinc(s_j) = 0$, por lo que el coste es infinito. Si dicho segmento apareciera en las soluciones de las demás conexiones, $coinc(s_j) > 0$, cuanto mayor sea el número de soluciones para la conexión j que utilicen el mismo segmento de partida, menor será el coste asignado al camino.

Por último, tampoco hay que olvidar el cambio a realizar en el punto 7, donde deben eliminarse los caminos que entren en conflicto con el camino seleccionado. En este caso se borrarán los caminos que, perte-

necientes a la misma red, aunque a distintas conexiones, no utilicen el mismo segmento de partida.

4.4.3 Resultados

En principio, el hecho de adaptar una herramienta de rutado ya existente al problema de los decodificadores, fue una solución adoptada para reducir la complejidad del estudio. Partir de cero en la elaboración de los algoritmos de rutado parecía mucho más complicado y tedioso que la adaptación de dichas herramientas. Sin embargo, debe señalarse que la gran cantidad de cambios introducidos junto con la comprensión del código original ha llevado, desgraciadamente *a posteriori*, a la conclusión de que el haber adaptado la herramienta ha sido precisamente la fuente de múltiples problemas que han complicado de forma notable el estudio en cuestión.

Para estudiar el efecto de los decodificadores en la rutabilidad, se han utilizado 10 de los circuitos más grandes del banco de pruebas del Centro Microelectrónico de Carolina del Norte [MCNC93] preparados por el autor de SEGA para su uso con dicha herramienta. Como medida de rutabilidad, se ha utilizado el número mínimo de líneas de segmentos necesarios por canal de rutado para conseguir rutar el circuito, lo que a partir de ahora se denominará W_{min}. Esta W_{min} se ha buscado para cada circuito y para las dos arquitecturas a comparar, una con decodificadores en los pines de salida y otra sin decodificadores.

En la tabla 4.1 se muestra el parámetro W_{min} para cada uno de estos circuitos y para cada arquitectura. La suma de dicho parámetro para una arquitectura determinada servirá como medida de la rutabilidad de la arquitectura. En este caso puede observarse que la reducción de la ru-

tabilidad es de un 13,2%. Aunque en principio puede parecer excesivo, debe de tenerse en cuenta que el uso de decodificadores no es un método para incrementar la rutabilidad de una FPGA, si no para reducir área. Por ello, el porcentaje anterior debe ser analizado conjuntamente con el ahorro en área del dispositivo. Como ya se verá en la sección de conclusiones, aún aumentando el número de líneas de segmentos por canal para contrarrestar el decremento de la rutabilidad, el ahorro en área sigue siendo significativo.

		sin decodificadores	con decodificadores
circuito	N° BL	W_{min}	W_{min}
alu4	1522	16	18
apex2	1878	20	21
apex4	1262	19	23
bigkey	1707	9	9
des	1591	11	13
diffeq	1497	10	11
dsip	1370	9	9
seq	1750	18	23
misex3	1397	17	21
ex5p	1064	16	19

Tabla 4.1: Rutabilidad

4.5 Retrasos

El cálculo de los retrasos producidos por las conexiones es fundamental para estimar la velocidad máxima de funcionamiento del circuito rutado. Idealmente debería usarse un simulador de circuitos como SPICE para obtener una buena estimación de los retrasos. Sin embargo esto resulta prohibitivo debido a la gran cantidad de redes que pueden existir en un circuito.

Por este motivo, algunos autores han desarrollado modelos para representar los transistores de paso mediante resistencias lineales y capa-

cidades [KHE93]. El modelo de un transistor de paso puede observarse en la figura 4.16, donde R_{paso} es la resistencia que ofrece el transistor al paso de la señal, y C_{dif} representa la capacidad que ofrece el drenador o la fuente de dicho transistor. Dicha capacidad es mayor si el transistor conduce, ya que el canal que se crea en el interior tiene una capacidad a puerta y a sustrato, sin embargo, como normalmente existen muchos más transistores cortados que conduciendo, el error que se comete al considerar sólo la capacidad de difusión es muy pequeño [BET99b]. En [KHE93] se utiliza el modelo de retrasos de Penfield-Rubinstein [RUB83] para determinar una cota máxima y mínima del retraso de la red RC. Otra alternativa, el modelo de Elmore [ELM48], es el más utilizado para el cálculo de retrasos en redes RC [CON96b] y consiste en lo siguiente:

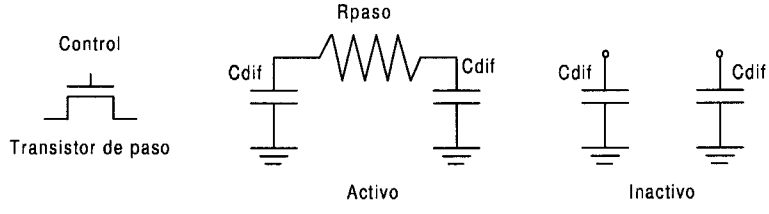


Figura 4.16: Modelo RC de un transistor de paso

$$T_{jk} = \sum_{i \in \text{camino j-k}} R_i \cdot C_{subred_i}$$

donde T_{jk} es el retraso desde la fuente j hasta el sumidero k, i es un índice que representa a los nodos de la red, y C_{subred_i} es la suma de capacidades de la subred conectada a R_i. Por ejemplo, para la red de la figura 4.17, el retraso desde la fuente hasta el sumidero 3 sería según la fórmula anterior:

$$T_3 = R_1 \cdot (C_1 + C_2 + C_3 + C_4) + R_2 \cdot (C_2 + C_3 + C_4) + R_3 \cdot C_3$$

Se ha demostrado [BOE93] [CON96a] que el modelo de Elmore ofrece

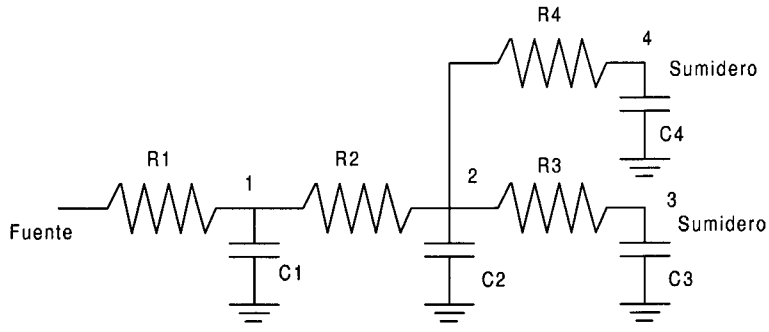


Figura 4.17: Ejemplo de red RC

una gran *fidelidad*, en el sentido de que si se ordenan las redes según el retraso calculado a partir del modelo de Elmore, existe una probabilidad muy alta de que así ocurra en la realidad. Sin embargo, dicho modelo no predice con exactitud su valor real. Por ello, para estudiar de forma exacta el efecto de los decodificadores en el retraso de las conexiones, se ha implementado el modelo de Elmore para hallar la red con mayor retraso. Una vez detectada, se procede a su simulación con SPICE para determinar con exactitud el valor de dicho retraso [BAE01].

Los valores de los elementos del modelo RC de los transistores de paso, dependen de la tecnología utilizada. Sin pérdida de generalidad, en la tabla 4.2, pueden observarse los valores utilizados para el cálculo de los retrasos. Los resultados obtenidos mediante la aplicación de este método a las soluciones de rutado obtenidas en la sección anterior, pueden verse en la tabla 4.3. De esta tabla se observa que el aumento en los retrasos no es significativo. Tan sólo un 2,92% en media, pero con una gran desviación típica.

Transistor activo	C_{dif}	$10fF$
	R_{paso}	$1K\Omega$
Transistor inactivo	C_{dif}	$10fF$
Buffer de salida	C_{fuente}	$100fF$
	R_{fuente}	100Ω
Buffer de entrada	$C_{sumidero}$	$20fF$

Tabla 4.2: Valores de los elementos del modelo RC

		sin decodificadores	con decodificadores
circuit	Nº BL	retraso (s)	retraso (s)
alu4	1522	1,42e-06	1,36e-06
apex2	1878	4,16e-07	4,29e-07
apex4	1262	1,30e-06	4,14e-07
bigkey	1707	2,10e-06	2,16e-06
des	1591	1,81e-06	1,41e-06
diffeq	1497	5,30e-07	1,10e-06
dsip	1370	1,37e-06	1,53e-06
seq	1750	4,14e-07	5,37e-07
misex3	1397	3,73e-07	1,23e-07
ex5p	1064	2,87e-06	3,89e-06

Tabla 4.3: Retrasos

4.6 Conclusiones

Como era de esperar, el uso de decodificadores en los pines de salida reduce la rutabilidad de la FPGA. Más concretamente, la reducción se ha estimado en un 13%, lo cual no es necesariamente malo: el ahorro en área permite aumentar los recursos de rutado para mejorar la rutabilidad y aún así mantener un ahorro considerable en área de silicio. Por ejemplo, supongamos una FPGA con 6 contextos, con bloques lógicos formados por 4 ELBs, y 28 líneas de segmentos por canal sin decodificadores. Si se utilizan decodificadores, el ahorro en área es aproximadamente un 20% pero la rutabilidad es menor que en la arquitectura sin decodificadores. Para aumentar dicha rutabilidad, se puede incrementar el número de líneas de segmentos por canal un 13% (de 28 a 32 líneas). Esto a su vez incrementa el área del dispositivo, disminuyendo

el ahorro en área a un 17%.

No obstante, de los resultados expuestos, está claro que el uso de decodificadores sólo es una técnica interesante cuando se usa para FPGAs multicontexto de, al menos, 3 contextos, por lo menos si la razón de su utilización es la del ahorro en área. En cuanto al aumento de los retrasos, se ha visto que no es muy significativo, aunque podría mejorarse utilizando técnicas específicas para su reducción, como las presentadas en [BET99a] (utilización de segmentos largos, inserción de búfferes en la arquitectura de rutado,...). Como el retraso de una red crece con el cuadrado de los transistores por los que pasa, al usar dichas técnicas, la diferencia en velocidad entre ambas arquitecturas podría reducirse considerablemente.

Capítulo 5

Aplicaciones a FIPSOC

En este capítulo se presentan los aspectos más relevantes del producto comercial resultado de esta tesis: la herramienta de síntesis automática *FLIPER* incluida en el entorno de desarrollo *FIPSOC CAE TOOLS*. Software para el desarrollo de aplicaciones en la familia FIPSOC, que puede conseguirse fácilmente a través de Internet en la dirección *http://www.sidsa.com/fipsoc*. Si bien las aportaciones de esta tesis son originales del autor, el desarrollo final de la herramienta es un trabajo realizado por el Grupo de Tecnología Electrónica del Departamento de Ingeniería Electrónica de la Universidad de Sevilla y forma parte del proyecto ESPRIT número 21625.

5.1 Introducción

Los estudios presentados en capítulos anteriores se plantean a partir de las diferentes situaciones que han ido apareciendo a lo largo del desarrollo de las herramientas de síntesis automática para la FPGA de FIPSOC. Para facilitar la lectura de esta tesis, el autor ha optado por presentar los estudios por separado y sin dar una razón exacta de la motivación de estos. Para solventar esta carencia, este capítulo pretende

aclarar el por qué de este trabajo y cómo cada uno de los capítulos ha contribuido al desarrollo de la herramienta FLIPER [BAE99c], así como aportar una visión general de la herramienta comercial resultante.

El dispositivo FIPSOC se presentó en sus inicios como algo totalmente original, tanto desde el punto de vista físico o arquitectural como desde el punto de vista de las herramientas de diseño. La incorporación de un microprocesador y de una parte analógica programable a la FPGA junto con las posibles interrelaciones entre las tres partes, imponía serias restricciones al diseño de la FPGA dando lugar a la aparición de situaciones nunca antes tratadas en la literatura relativa al diseño de las herramientas CAD dedicadas a FPGAs. Cabe destacar, entre otras, la limitación de los recursos de rutado, muy por debajo de los utilizados en FPGAs comerciales del mismo tipo, y las distintas asimetrías presentes, como los diferentes tamaños de los canales de rutado horizontales y verticales. Como suele ocurrir en estos casos, el programador debe atenerse a la arquitectura elegida, y desarrollar los algoritmos necesarios para su programación.

Una característica importante del dispositivo es su reconfigurabilidad dinámica y su capacidad multicontexto, que, como ya se ha dicho en el capítulo anterior, posibilita la reprogramación de parte o la totalidad del dispositivo mientras éste sigue activo y permite el almacenamiento de varias configuraciones (2 contextos) que serán activadas mediante una señal.

El circuito posee otras muchas características que no serán detalladas en esta tesis debido a que no presentan mayor interés para el trabajo aquí expuesto. Para mayor información ver [SID99a] y [SID99b].

5.2 Interfaz gráfica

La herramienta desarrollada e integrada dentro del sistema "FIPSOC CAE tools", presenta la interfaz mostrada en la figura 5.1. Ésta permite tanto la ejecución paso a paso por las distintas etapas, como el modo totalmente automático. La selección de uno u otro método dependerá de la configuración del usuario.

La interfaz presenta 4 zonas bien diferenciadas, en la zona central aparece la ventana *General Info* que constituye la fuente principal de información para el usuario e indica de forma somera los resultados obtenidos por las diferentes etapas. Dependiendo de la etapa en ejecución, aparecerán otras ventanas que serán de ayuda para el usuario. En la zona de la derecha, las barras de progreso indican al usuario el estado de ejecución de las distintas etapas mientras que los botones permiten un acceso rápido a las funciones más utilizadas. La parte superior permite acceder mediante la barra de menús a todas las funciones del programa. Por último, la zona inferior presenta una barra de estado que irá mostrando los mensajes más importantes a lo largo del diseño.

Uno de los requisitos más importantes de una herramienta de desarrollo es su facilidad de utilización. En cada etapa, el usuario es guiado debido a que sólo podrá realizar unas funciones determinadas, convenientemente señaladas debido a la activación o desactivación de los botones y menús, lo cual facilita el uso de la herramienta para usuarios inexpertos. También permite al usuario avanzado el acceso a funciones de bajo nivel como es el cambio de parámetros de los algoritmos mediante el menú correspondiente.

5.3 Empaquetado

La primera etapa de la herramienta consistirá en empaquetar los elementos de entrada como LUTs y Flip-Flops en bloques lógicos. Como ya se ha dicho anteriormente, los bloques lógicos de la FPGA de FIPSOC están formados por 4 LUTs de 4 entradas, 4 Flip-Flops y un bloque de rutado interno que permite la interconexión entre parte combinacional y parte secuencial. Este último bloque, de una gran versatilidad, admite múltiples configuraciones posibilitando una gran conectividad. Como característica adicional, cabe destacar el hecho de que las LUTs comparten entradas, como ya se describió en el capítulo 2.

El algoritmo empleado para el empaquetado varía muy poco del presentado en el capítulo 2. El resultado de aplicar el algoritmo a un circuito puede verse en la figura 5.1, donde aparecen algunos parámetros del circuito de entrada (número de redes, número de LUTs, etc...) así como de la solución hallada (número de bloques lógicos, ocupación de los distintos elementos, etc...).

5.4 Colocación

Una vez terminado el empaquetado, la etapa de colocación distribuirá los bloques lógicos dentro de la FPGA. En la figura 5.2 puede verse el resultado final de dicha etapa para un circuito. En esta figura puede observarse el carácter rectangular de la FPGA así como la existencia en la parte inferior derecha de unos puntos especiales, los puertos, que permiten la conexión de la FPGA con el microprocesador o con la parte analógica.

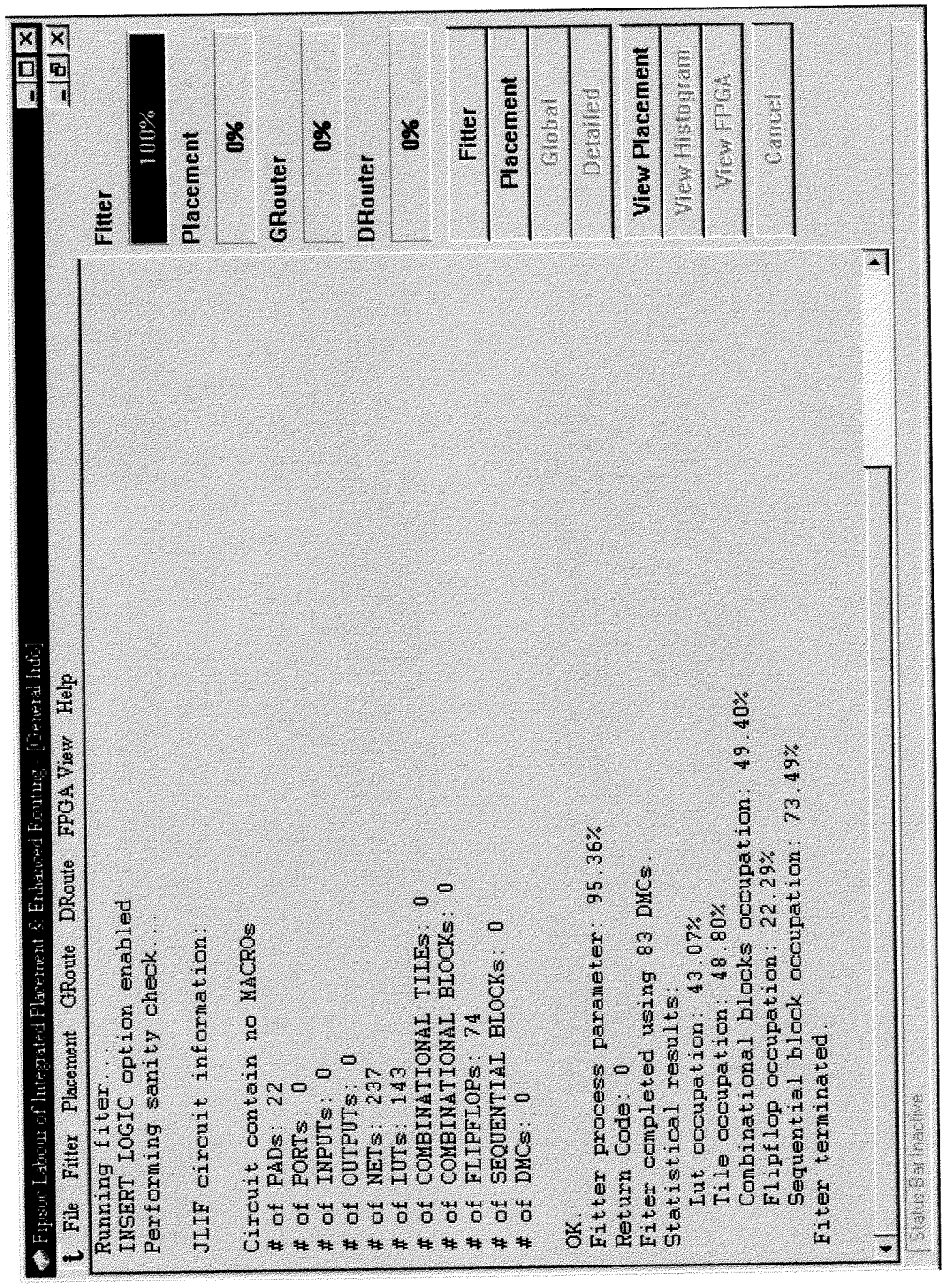


Figura 5.1: Interfaz general

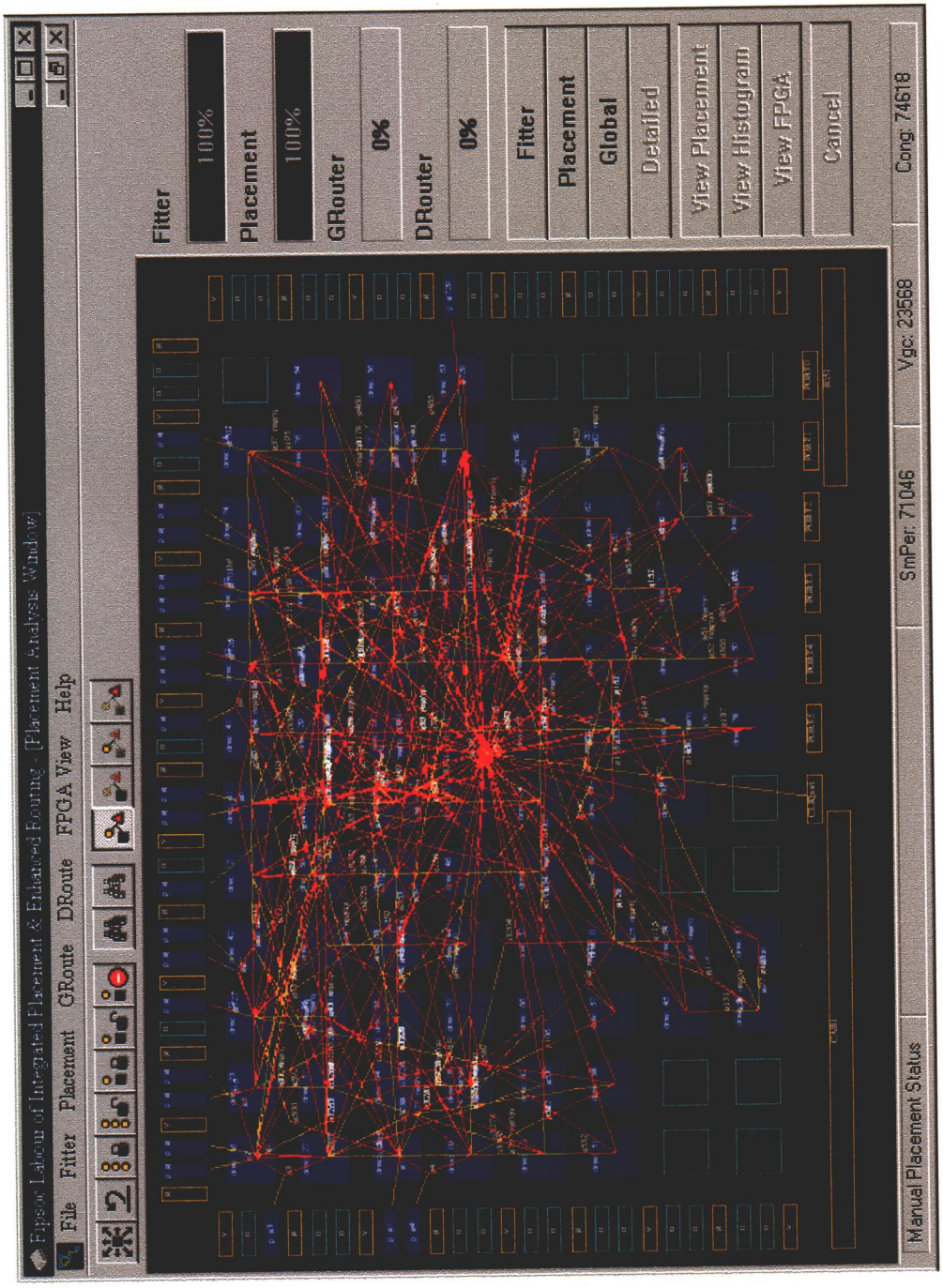


Figura 5.2: Ventana de colocación

La barra de botones que aparece en la zona superior de la ventana, permite al usuario mover los bloques lógicos o PADs manualmente, o bloquear la posición de algún bloque lógico o PAD para la ejecución posterior del algoritmo de colocación sin que éste los mueva a otra posición, algo muy importante para los PADs si se está desarrollando pruebas para un dispositivo y la placa de circuito impreso ya está realizada. También aparece toda una serie de funciones adicionales que ayudan al usuario en el diseño del dispositivo, permitiéndole incluso la utilización de la reconfigurabilidad parcial del dispositivo.

Visualizar el circuito ya colocado, o cambiar manualmente la propiedad de algún elemento es tan sólo una opción, y no es necesario para el desarrollo completo, por lo que es decisión del usuario acceder o no a esas funciones, facilitando así el uso de la herramienta para usuarios inexpertos.

Además de incluir una versión adaptada a FIPSOC del algoritmo de colocación VGC descrito en el capítulo 3, se ha optado por integrar dos algoritmos más: de retrasos y de congestión, basados en trabajos ya publicados. De esta manera, se ofrecen varias posibilidades al usuario avanzado en el caso de encontrarse con un circuito difícil de rutar.

5.5 Rutado

Tras el proceso de colocación, la etapa de rutado seleccionará segmentos e interruptores programables para realizar las conexiones entre los bloques lógicos. Tal como se describió en el capítulo 4, se ha dividido el proceso de rutado en dos etapas: rutado global y detallado. La primera se encarga de seleccionar para cada conexión los canales de rutado

por donde pasarán las redes. Para su programación se ha realizado una adaptación del algoritmo presentado en [BAE97] para la minimización de la congestión de los canales de rutado.

La segunda etapa, el rutado detallado, utiliza el algoritmo descrito en el capítulo 4 aunque con algunas modificaciones. El problema principal al dividir el proceso de rutado en dos etapas, es el de modelar correctamente los recursos de rutado en la etapa de rutado global. En realidad, el único cometido del rutado global es el de restringir el espacio de búsqueda de posibles soluciones para facilitar la tarea a la etapa siguiente. La FPGA de FIPSOC es tan asimétrica, que para asegurar un buen funcionamiento de la etapa de rutado global, el modelo de los recursos de rutado debe ser muy completo (mayor complejidad y por lo tanto mayor tiempo de ejecución), con lo que el problema de rutado global se convierte casi en el del rutado detallado. Las únicas soluciones encontradas son 3:

1. Utilizar un modelo complejo de la arquitectura de rutado de la FPGA para el correcto funcionamiento del algoritmo de rutado global.

2. Utilizar un modelo simplificado pero permitir que el rutado detallado pueda explorar el espacio de soluciones completo cuando exista alguna conexión que no se pueda rutar.

3. Realizar el proceso de rutado en una sola etapa.

Como ya se dijo en el capítulo anterior, no se ha podido encontrar un algoritmo capaz de resolver el rutado en una sola etapa, por lo que la tercera posibilidad quedaba descartada. La primera implica un alto tiempo de ejecución que además es independiente de la complejidad del circuito a rutar, algo muy negativo desde el punto de vista del usuario final. La

segunda opción disminuye el tiempo de ejecución reduciendo la complejidad del modelo de la arquitectura de rutado. Consecuencia de esto, es una mala estimación de la congestión en los canales de rutado por lo que el proceso de rutado global no restringe adecuadamente el espacio de soluciones y el rutado detallado no consigue rutar el circuito si no se le permite explorar un espacio de búsqueda más amplio para las conexiones *difíciles*.

Está claro que la mejor de las dos en cuanto a tiempo de ejecución es la segunda aunque la complejidad en cuanto a programación se refiere es muy superior. Experimentos previos demostraron que en los casos en los que no se encontraba una solución al problema de rutado, más del 90% de las conexiones podían ser rutadas a partir de la solución generada por el rutado global. De alguna forma, sólo había que buscar una solución para el 10% restante, es decir, sólo había que aumentar el espacio de búsqueda para un 10% de las conexiones.

La solución adoptada en FLIPER es la siguiente: la etapa de rutado global trata de restringir el espacio de búsqueda mediante un modelo simplificado y genera una primera aproximación que será utilizada por el proceso de rutado detallado. En principio, el algoritmo utilizado no difiere del presentado en el capítulo 4 y si se alcanza una solución el proceso termina aquí. Sin embargo, si no se consiguen rutar algunas conexiones, se emplea el "algoritmo del laberinto" utilizado en VPR para encontrar una solución a estas conexiones aumentando el espacio de búsqueda y sin modificar el rutado de otras redes. Como ya se vio en el capítulo anterior, el algoritmo utilizado en VPR es muy inestable cuando se usan decodificadores a la salida de los bloques lógicos, sin embargo, tras múltiples pruebas se comprobó que esa inestabilidad desaparece si

el número de conexiones a rutar es pequeño y, sobre todo, si el espacio de búsqueda se reduce como consecuencia de que algunas conexiones ya están rutadas. Por este mismo motivo, el aumento del tiempo de ejecución no es muy elevado, y además sólo se produce cuando la dificultad del circuito así lo requiere.

Por último, en cuanto a la interfaz gráfica se refiere, en la figura 5.3 puede verse la estimación de la congestión de los canales de rutado realizada por la etapa de rutado global, mientras que en las figuras 5.4 y 5.5 puede verse el resultado final de la etapa de rutado detallado. Como ayuda para usuarios de alto nivel, el programa ofrece ciertas posibilidades de análisis, como la búsqueda de señales y su posterior visualización en pantalla; así como la posibilidad de ver la configuración interna de los bloques lógicos y transistores de paso.

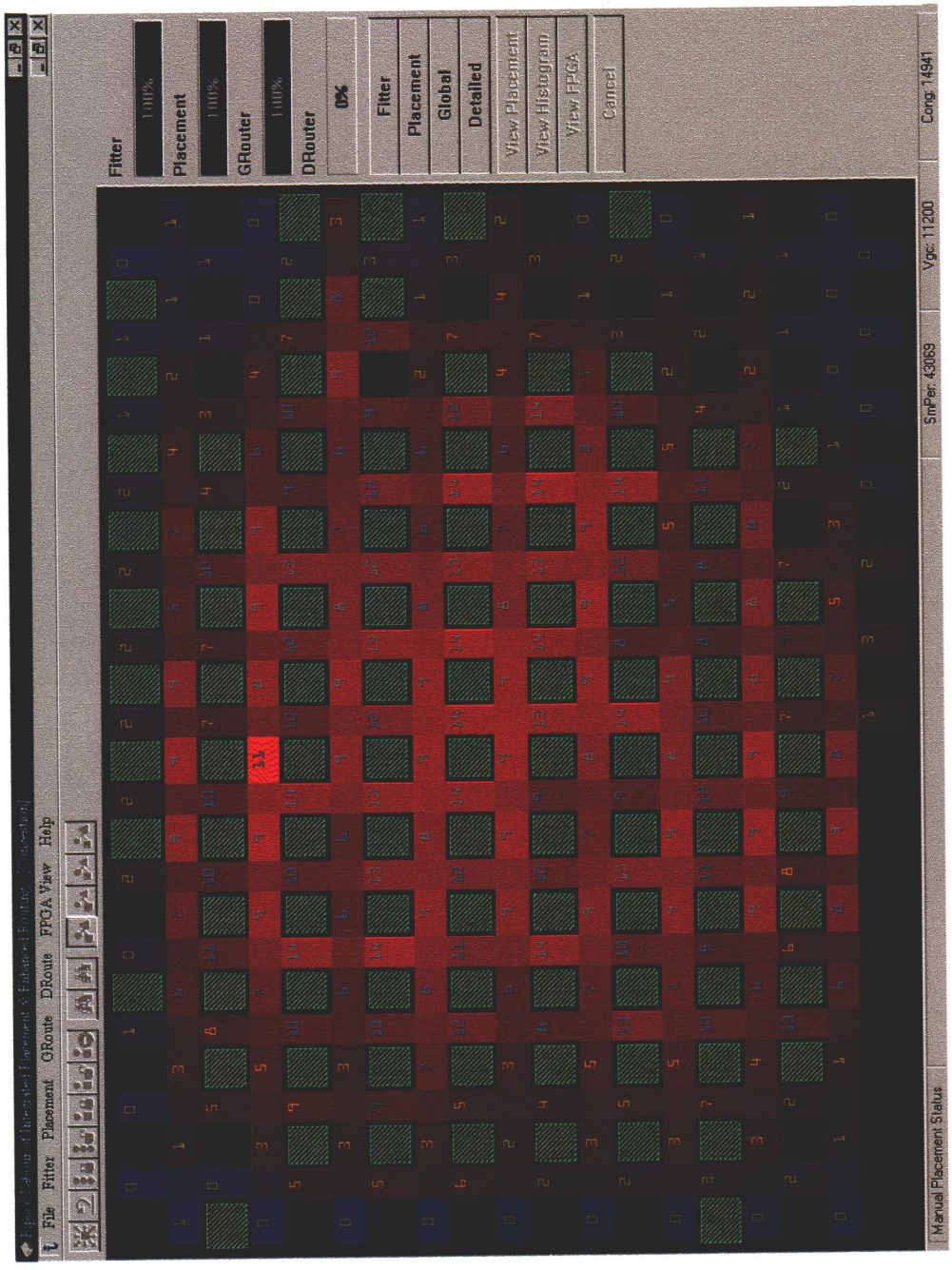


Figura 5.3: Estimación de la congestión en los canales de rutado

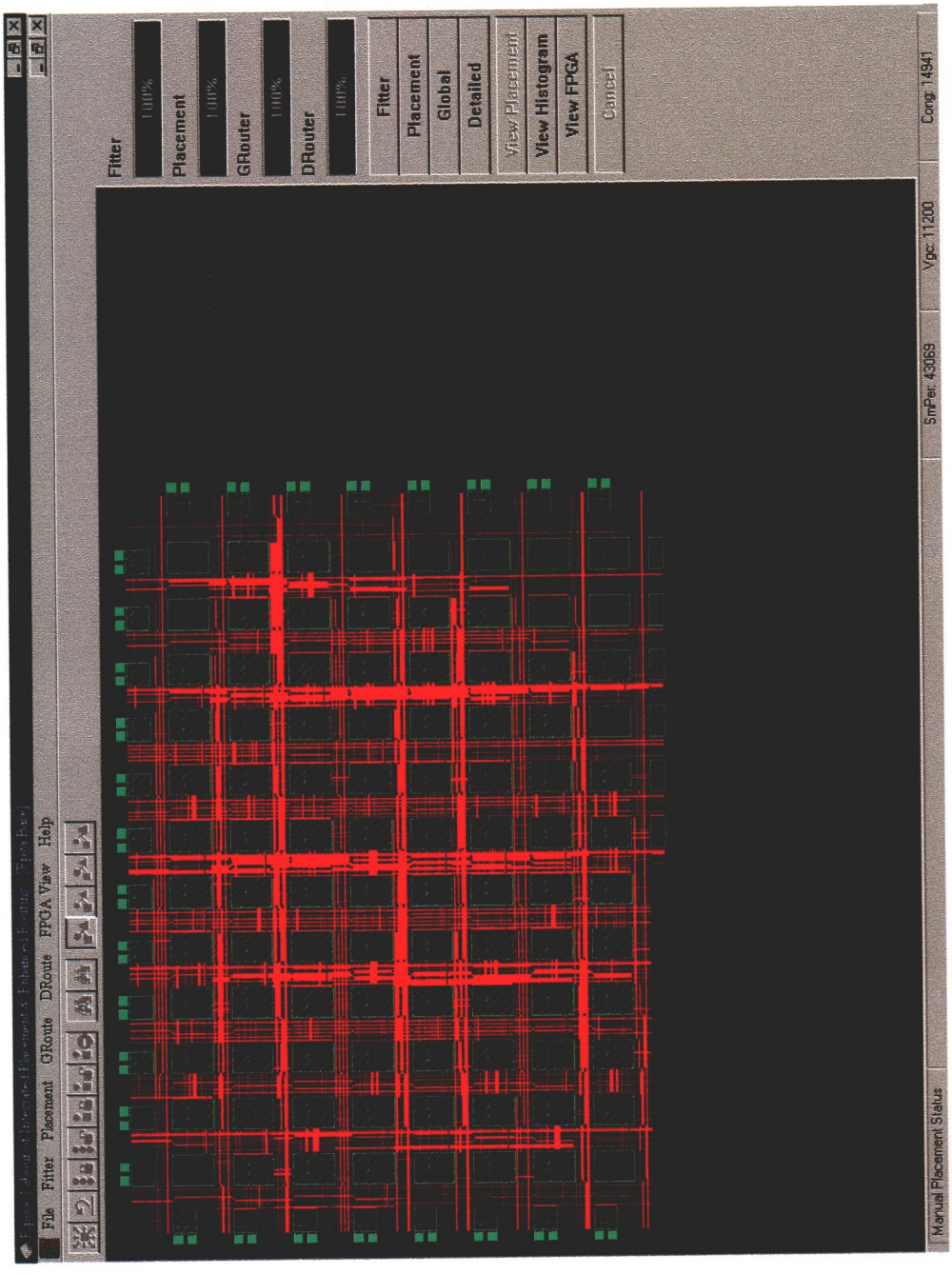


Figura 5.4: Resultado de la etapa de rutado detallado

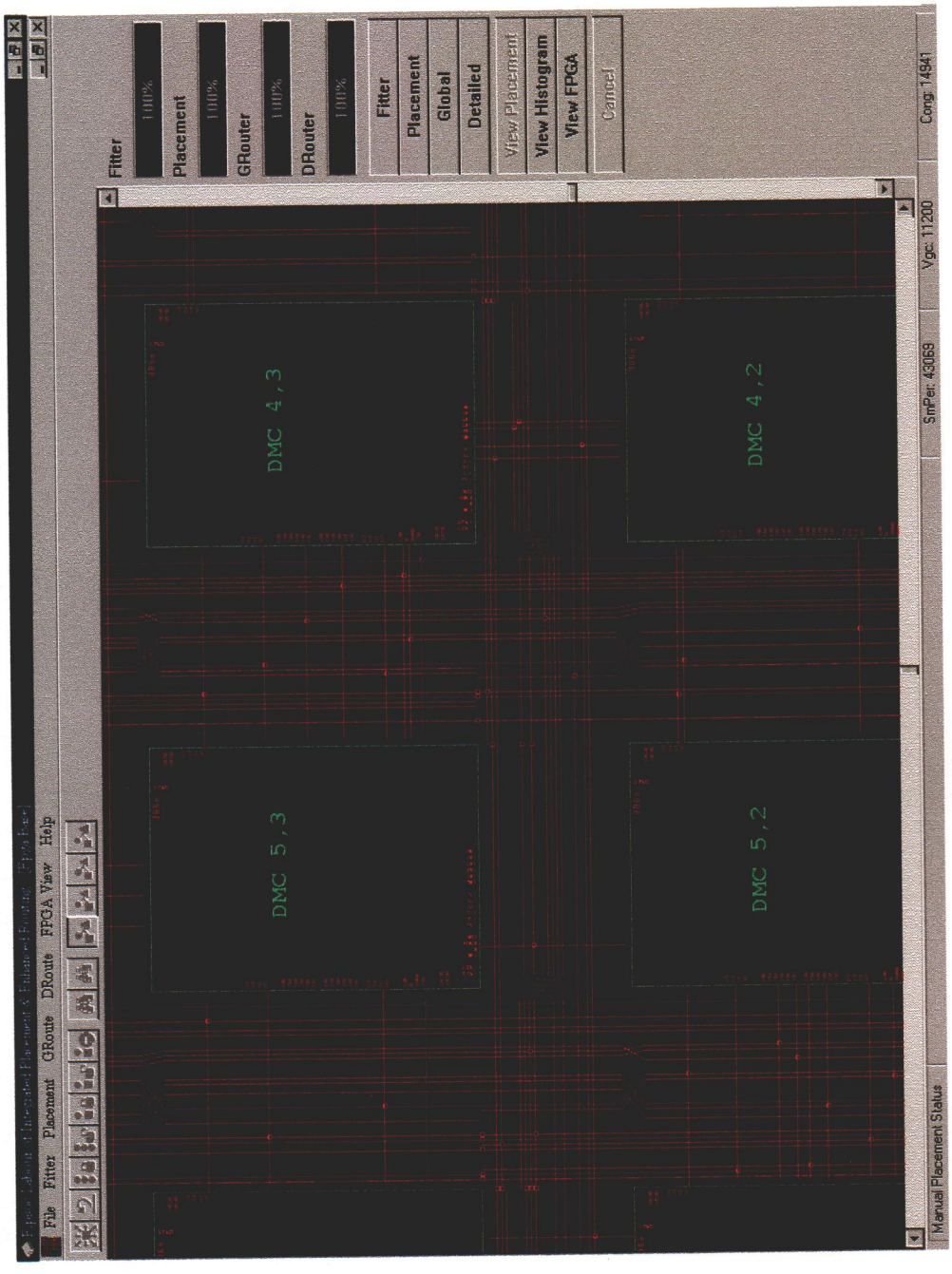


Figura 5.5: Detalle de la solución del rutado detallado

Capítulo 6

Conclusiones

A lo largo de los capítulos anteriores, han sido presentados una gran variedad de aportaciones y estudios, sobre los dispositivos lógicos programables. A grandes rasgos, dichas aportaciones pueden ser divididas en dos grupos:

1. Algorítmicas: relacionadas con las herramientas de programación del dispositivo.

2. Arquitecturales: relacionadas con el diseño físico del dispositivo.

En el primer grupo, cabe destacar, el desarrollo de un nuevo algoritmo de colocación, y una nueva herramienta de rutado específicamente diseñada para arquitecturas con decodificadores en los pines de salida de los bloques lógicos. En el segundo, tienen especial relevancia el estudio realizado sobre la compartición de los pines de entrada de los bloques lógicos, así como el estudio del uso de decodificadores, para conectar los pines de salida de los bloques lógicos, con los canales de rutado.

La clara interrelación entre estos dos grupos, hace imposible el tratamiento individual de cada estudio. Por este motivo, a continuación se procederá a resumir las conclusiones más relevantes de cada capítulo.

En el capítulo 2, se ha mostrado que, al compartir algunos pines de entrada, entre LUTs de un mismo bloque lógico, puede incrementarse la rutabilidad de los circuitos, sin por ello disminuir la eficiencia en área del dispositivo. Concretamente, se ha observado que si las LUTs de un bloque lógico comparten 1 pin, el número de conexiones a rutar se reduce en un 13%, todo ello sin aumentar el área ocupada por el circuito. Se sabe además, que este resultado es una cota inferior por dos motivos. El primero, por la no utilización de los modos *MACRO*, los cuales reducen considerablemente las conexiones y facilitan el empaquetamiento. El segundo, si las conexiones se reducen en un 13%, se incrementa la rutabilidad, por lo que no son necesarios tantos recursos de rutado, es decir, pueden reducirse, disminuyendo el área de silicio. Este resultado es fácilmente generalizable a cualquier arquitectura con bloques lógicos complejos, como la VIRTEX de Xilinx [XIL99].

En el capítulo 3, se ha presentado un nuevo algoritmo de colocación que mejora considerablemente los resultados obtenidos por el, hasta hoy, mejor programa de colocación: VPR [BET97a]. En particular, esta mejora se produce para FPGAs con arquitecturas formadas por segmentos con longitud mayor que la unidad, es decir, arquitecturas más cercanas a las de las FPGAs comerciales. Los resultados demuestran una mejora de un 17%, para estas arquitecturas. El algoritmo presentado permite, además, la definición de cualquier tipo de asimetría, problema común de los SOCs.

En el capítulo 4, han sido analizados los efectos producidos por el uso de decodificadores en los pines de salida de los bloques lógicos. En los pines de entrada, prácticamente todas las FPGAs comerciales usan decodifi-

cadores, debido al ahorro en memoria de configuración. Las entradas de los bloques lógicos, suelen conectarse a una única salida, por lo que en este caso, el uso de decodificadores no plantea problema alguno en el diseño de la herramienta automática de rutado. Sin embargo, no ocurre así en los pines de salida. Estos sí pueden estar conectados a varias entradas. Por este motivo, al usar decodificadores en los pines de salida, se limita a uno el número de segmentos a los cuales puede conectarse el pin al mismo tiempo, lo que plantea serios problemas a la algorítmica asociada al programa de rutado. Dicho problema ha sido resuelto eficazmente, sin aumentar el tiempo de ejecución. Se ha desarrollado una herramienta capaz de rutar circuitos en una arquitectura con decodificadores en los pines de salida. Además, se han estudiado los efectos producidos por estos en el área de silicio y en los retrasos de las conexiones. Los resultados obtenidos, demuestran que, la aplicación de esta técnica, puede reducir el área de silicio ocupada por el dispositivo, sin disminuir la rutabilidad ni aumentar los retrasos de las conexiones. En particular, para una FPGA de 6 contextos, el área puede reducirse en un 17%.

Por último, en el capítulo 5, se ha presentado FLIPER, la herramienta comercial utilizada en el proceso de programación de la FPGA de FIP-SOC, resultado de la aplicación del trabajo realizado en esta tesis.

Bibliografía

[ACT99a] Actel, "FPGA Data Book and Design Guide", Data Book de Actel, 1999.

[ACT99b] Actel, "ProASIC 500K Family Data Sheet (Advance V3)", Data Sheet, Diciembre de 1999.

[AHM00] E. Ahmed y J. Rose, "The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density", Proceedings of the 2000 International Symposium on Field-Programmable Gate Arrays, FPGA'2000, Monterey, Estados Unidos, Febrero de 2000, pp. 3-12.

[ALE94] M. Alexander, J. Cohoon, J. Ganley y G. Robins, "An Architecture-Independent Approach to FPGA Routing Based on Multi-Weighted Graphs", Proceedings of the 1994 European Design Automation Conference, EDAC'94, Grenoble, Francia, Septiembre de 1994, pp. 259-264.

[ALE96] M. J. Alexander y G. Robins, "New Performance-Driven FPGA Routing Algorithms", IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Diciembre de 1996, pp. 1505-1517.

[ALT99] Altera, "Data Book", Data Book de Altera, 1999.

[ATM99] Atmel, "AT94K FPSLIC Advance Information Summary", Data Sheet, diciembre de 1999.

[BAE97] V. Baena-Lecuyer, M.A. Aguirre, A. Torralba, L.G. Franquelo y J. Faura, "RAISE: A detailed Routing Algorithm for Field-Programmable Gate Arrays", Proceedings of the 1997 International Conference on Design of Circuits and Integrated Systems, DCIS'97, Sevilla, España, Noviembre de 1997, pp. 421-424.

[BAE98] V. Baena-Lecuyer, M.A. Aguirre, A. Torralba, L. G. Franquelo y J. Faura, "RAISE: A Detailed Routing Algorithm for SRAM based Field-Programmable Gate Arrays using Multiplexed Switches", Proceedings of the 1998 International Symposium on Circuits and Systems, ISCAS'98, Monterey, Estados Unidos, Mayo de 1998, pp. 430-433.

[BAE99a] V. Baena-Lecuyer, M.A. Aguirre, A. Torralba, L. G. Franquelo y J. Faura, "Using Decoders for the Connecting Matrix in Multicontext FPGA's: Area Reduction versus their Effect on Routability", Proceedings of the 1999 International Conference on Mixed Design of Integrated Circuits and Systems, MIXDES'99, Crakovia, Polonia, Junio de 1999, pp 145-148.

[BAE99b] V. Baena-Lecuyer, M.A. Aguirre, A. Torralba, L. G. Franquelo y J. Faura, "Decoder-Driven Switching Matrices in Multicontext FPGA's: Area Reduction and their Effects on Routability", Proceedings of the 1999 International Symposium on Circuits and Systems, ISCAS'99, Orlando, Estados Unidos, Mayo de 1999, pp 463-466.

[BAE99c] V. Baena-Lecuyer, M.A. Aguirre, A. Torralba, L.G. Franquelo y J. Faura, "FLIPER: Physical Implementation CAE flow for a

System On Chip", Proceedings of the 1999 International Conference on Design of Circuits and Integrated Systems, DCIS'99, Palma de Mallorca, España, Noviembre de 1999, pp. 593-598.

[BAE00] V. Baena-Lecuyer, M.A. Aguirre, A. Torralba y L.G. Franquelo, "Evaluating Placement for Asymmetric FPGA's and PSOC Devices using a Most Direct Path Approach", Proceedings of the 2000 International Conference on Design of Circuits and Integrated Systems, DCIS'2000, Montpellier, Francia, Noviembre de 2000, pp. 83-87.

[BAE01] V. Baena-Lecuyer, M.A. Aguirre, A. Torralba y L.G. Franquelo, "Decoder-Driven Switching Matrices in Multicontext FPGAs: Area, Routability and Speed", aceptado para el International Symposium on Field-Programmable Gate Arrays, FPGA'2001, Monterey, Estados Unidos, Febrero de 2001.

[BET97a] V. Betz y J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research", Proceedings of the 1997 International Workshop on Field-Programmable Logic and Applications, FPL'97, Londres, Reino Unido, Septiembre de 1997, pp. 213-22.

[BET97b] V. Betz y J. Rose, "Cluster-Based Logic Blocks for FPGAs vs. Input Sharing and Size", Proceedings of the 1997 Custom Integrated Circuits Conference, CICC'97, Santa Clara, Estados Unidos, mayo de 1997, pp. 551-554.

[BET98] V. Betz, "VPack and VPR User´s Manual", 1998. Puede ser descargado en:
http://www.eecg.toronto.edu/~jayar/software/software.html.

[BET99a] V. Betz y J. Rose, "FPGA Routing Architecture: Segmentation and Buffering to Optimize Speed and Density", Proceedings of the 1999 International Symposium on Field-Programmable Gate

Arrays, FPGA'99, Monterey, Estados Unidos, Febrero de 1999, pp. 140-149.

[BET99b] Vaughn Betz, Jonathan Rose y Alexander Marquadt, "Architecture and CAD for Deep-Submicron FPGAs", Kluwer Academic Publishers, ISBN 0-7923-8460-1,1999.

[BOE93] K. Boese, A. Kahng, B. McCoy y G. Robbins, "Fidelity and Near-Optimality of Elmore Based Routing Constructions", Proceedings of the 1993 International Conference on Computer Design, ICCD'93, Cambridge, Estados Unidos, Octubre de 1993, pp. 81-84.

[BRO92a] S.D. Brown "Routing Algorithms and Architectures for Field Programmable Gate Arrays", Tesis doctoral, Department of Electrical Engineering, Universidad de Toronto, Enero de 1992.

[BRO92b] Stephen D. Brown, Robert J. Francis, Jonathan Rose y Zvonko G. Vranesic, "Field Programmable Gate Arrays", Kluwer Academic Publishers, ISBN 0-7923-9248-5, 1992.

[BRO92c] S. Brown, J. Rose, Z. G. Vranesic, "A detailed Router for Field-Programmable Gate Arrays", IEEE Transactions on Computer Aided Design, Mayo de 1992, pp. 620-628.

[CHE94] C.-L. E. Cheng, "RISA: Accurate and Efficient Placement Routability Modeling", Proceedings of the 1994 International Conference on Computer Aided Design, ICCAD'94, San Jose, Estados Unidos, Noviembre de 1994, pp. 690-695.

[CON96a] J. Cong y L. He, "Optimal Wiresizing for Interconnects with Multiple Sources", ACM Transactions on Design Automation of Electronic Systems, Vol. 1, N°4, Octubre de 1996, pp. 478-511.

[CON96b] J. Cong, L. He, C. Koh y P. Madden, "Performance Optimization of VLSI Interconnect Layout", Integration, The VLSI Journal, Vol. 21, 1996, pp. 1-94.

[CON96c] J. Cong, J. Peck y Y. Ding, "RASP: A General Logic Synthesis System for SRAM-based FPGAs", Proceedings of the 1996 International Symposium on Field-Programmables Gate Arrays, FPGA'96, Monterey, Estados Unidos, Febrero de 1996, pp. 137-143.

[DEH96] André DeHon, "Reconfigurable Architectures for General-Purpose Computing", Tesis doctoral, Massachussets Institute of Technology, MIT, Septiembre de 1996.

[EBE95] C. Ebeling, L. McMurchie, S.A. Hauck y S. Burns, "Placement and Routing Tools for the Tryptich FPGA", IEEE Transactions On VLSI, diciembre de 1995, pp. 473-482.

[ELM48] W. Elmore, "The transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers", Journal of Applied Physics, Enero de 1948, pp. 55-63.

[FAR94] Amir H. Farrahi y Majid Sarrafzadeh, "Complexity of the Lookup-Table Minimization Problem for FPGA Technology Mapping", IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Noviembre de 1994, Vol. 13, No 11, pp. 1319-1332.

[FAU97] J. Faura, J.M. Moreno, C. Horton, P.V. Duong, M.A. Aguirre and J.M. Insenser. "Multicontext Dynamic Reconfiguration and Real Time Probing on a Novel Mixed Signal Programmable Device with On–Chip Microprocessor", Proceedings of the 1997 International Workshop on Field-Programmable Logic and Applications, FPL'97, Londres, Reino Unido, Septiembre de 1997, pp. 1-10.

[FRA91] , R. Francis, J. Rose y Z. Vranesic. "Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs", Proceedings of the 1991 Design Automation Conference, DAC'91, San Francisco, Estados Unidos, Junio de 1991, pp. 227-233.

[FRA92] J. Frankle, "Iterative and Adaptive Slack Allocation for Performance Driven-Layout and FPGA Routing", Proceedings of the 1992 Design Automation Conference, DAC'92, Anaheim, Estados Unidos, Junio de 1992, pp. 536-542.

[HE93] J. He y J. Rose, "Advantages of Heterogenous Logic Block Architectures for FPGAs", Proceedings of the 1993 Custom Integrated Circuits Conference, CICC'93, San Diego, Estados Unidos, Mayo de 1993, pp. 7.4.1 - 7.4.5.

[HIL93] Dwight Hill y Nam-Sung Woo, "The Benefits of Flexibility in Lookup Table-Based FPGA's", IEEE Transactions on Computer Aided Design of Integrated Circuits an Systems, Febrero de 1993, Vol. 12, No 2, pp. 349-353.

[HUA86] M. Huang, F. Romeo y A. Sangiovanni-Vicentelli, "An Efficient General Cooling Schedule for Simulated Annealing", Proceedings of the 1986 International Conference on Computer Aided Design, ICCAD'86, Santa Clara, Estados Unidos, Noviembre de 1986, pp. 381-384.

[HWA94] Ting-Ting Hwang, Robert Michael Owens y Mary Jane Irwin, "Logic Synthesis for Field-Programmable Gate Arrays", IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Octubre de 1994, Vol. 13, No 10, pp. 1280-1287.

[KAR91] K. Karplus, "Xmap: A Technology Mapper for a Table-Lookup Field Programmable Gate Array", Proceedings of the 1991 Design

Automation Conference, DAC'91, San Francisco, Estados Unidos, Junio de 1991, pp. 240-243.

[KHE93] M. Khellah, S. Brown y Z. Vranesic, "Modelling Routing Delays in SRAM-Based FPGAs", Proceedings of the 1993 Canadian Conference on VLSI, FPD'93, Banff, Canadá, Noviembre de 1993, pp. 6B.13-6B.18.

[KIR83] S. Kirkpatrick, C. Gelatt y M. Vecchi, "Optimization by Simulated Annealing", Science, Mayo de 1983, pp. 671-680.

[KLE91] J. Kleinhans, G. Sigl, F. Johannes y K. Antreich, "Gordian: VLSI Placement by Quadratic Programming and Slicing Optimization", IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Marzo de 1991, pp. 356-365.

[LAM88] J. Lam y J. Delosme, "Performance of a New Annealing Schedule", Proceedings of the 1988 Conference on Design Automation, DAC'88, Anaheim, Estados Unidos, Junio de 1988, pp. 306-311.

[LEE97] Y.-S. Lee y A. Wu, "A Performance and Routability Driven Router for FPGAs Considering Path Delays", IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Febrero de 1997, pp. 178-185

[LEM93] G. Lemieux y S. Brown, "A Detailed Router for Allocating Wire Segments in FPGAs", Proceedings of the 1993 Physical Design Workshop, Lake Arrowhead, Estados Unidos, Abril de 1993, pp. 215-226.

[LUC00] Lucent Technologies, "ORCA Series 4 Field-Programmable Gate Arrays", Data Sheet, Agosto de 2000.

[LYS93] P. Lysaght y J. Dunlop, "Dynamic reconfiguration of FPGAs", Proceedings of the 1993 International Workshop on Field-Programmable Logic and Applications, FPL'93, Oxford, Reino Unido, Septiembre de 1993, pp.82-94.

[MCNC93] Banco de circuitos de prueba LGSynth93 del Centro Microelectrónico de Carolina del Norte (MCNC93).

[NAG95] S. Nag y R. Rutenbar, "Performance-Driven Simultaneous Place and Route for Island-Style FPGAs", Proceedings of the 1995 International Conference on Computer Aided Design, ICCAD'95, San Jose, Estados Unidos, Noviembre de 1995, pp. 332-338.

[NAG98] S. Nag y R. Rutenbar, "Performance-Driven Simultaneous Place and Route for Island-Style FPGAs", IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Junio de 1998, pp. 499-518.

[ROS90] J. Rose, R. Francis, D. Lewis y P. Chow, "Architecture of Field Programmable Gate Arrays: The Effect of Logic Block Functionality on Area efficiency", IEEE Journal of Solid-State Circuit, Vol 25, N° 5, Octubre de 1990, pp. 1217-1225.

[ROS91] J. Rose y S. Brown, "Flexibility of Interconnection Structures for Field Programmable Gate Arrays", IEEE Journal of Solid State Circuit, Marzo de 1991, pp. 227-282.

[ROS93] J. Rose, A. El Gamal y A. Sangiovanni-Vincentelli, "Architecture for Field-Programmable Gate Arrays", Proceedings of the IEEE, Julio de 1993, pp. 1013-1029.

[RUB83] J. Rubinstein, P. Penfield y M. Horowitz, "Signal Delay in RC Tree Networks", IEEE Transactions on Computer Aided Design, 1983, pp. 202-211.

[SEC88] C. Sechen, "VLSI Placement and Global Routing using Simulated Annealing", Kluwer Academic Publishers, ISBN 0-89838-281-5, 1988.

[SEN92] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, Alberto Sangiovanni-Vincentelli, "SYS: a System for Synthesis of Sequential Circuits", Memorandum No. UCB/ERL M92/41, Electronic Research Laboratory, University of California, Berkeley, 4 de Mayo de 1992.

[SID99a] SIDSA, "FIPSOC User Manual", Manual de usuario de la fpga FIPSOC, 1999.

[SID99b] SIDSA, "FIPSOC Datasheet", Hoja de características del dispositivo FIPSOC, 1999.

[SWA90] W. Swartz y C. Sechen, "New Algorithms for the Placement and Routing of Macro Cells", Proceedings of the 1990 International Conference on Computer Aided Design, ICCAD'90, Santa Clara, Estados Unidos, 1990, pp. 336-339.

[TAU95] E. Tau, D. Chen, I. Eslick, J. Brown y A. DeHon, "A First Generation DPGA Implementation", Proceedings of the 1995 Canadian Workshop on Field-Programmable Devices, FPD'95, Montreal, Canadá, Mayo de 1995, pp. 138-143.

[THA97] S. Thakur, Y.-W. Chang, D.F. Wong y S. Muthukrishnan, "Algorithms for an FPGA Switch Module Routing Problem with Application to Global Routing", IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Enero de 1997, pp. 32-46.

[TSE92] B. Tseng, J. Rose y S. Brown, "Using Architectural and CAD Interactions to improve FPGA Routing Architectures", Proceedings of the 1992 International Conference on Computer Design, ICCD'92, Cambridge, Estados Unidos, Octubre de 1992, pp. 99-104.

[WU97] Y.-L. Wu, M. Marek-Sadowska, "Routing for Array-Type FPGA´s", IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Mayo de 1997, pp. 506-518.

[XIL97] XC6200 Field Programmable Gate Arrays. Xilinx, 1997.

[XIL99] Xilinx, "The Programmable Logic Data Book", Data Book de Xilinx, 1999.

[YAN91] S. Yang, "Logic Synthesis and Optimization Benchmarks, version 3.0", Tech. Report, Microelectronic Center of North Carolina, 1991.