UNIVERSIDAD DE SEVILLA

ESCUELA SUPERIOR DE INGENIEROS TELECOMUNICACIONES

DEPARTAMENTO DE ELECTRONICA

HERRAMIENTA DE MAPEO LÓGICO SOBRE UNA TECNOLOGÍA FPGA DINÁMICAMENTE RECONFIGURABLE. FIPSOC.

PROYECTO

Que para obtener el título de

Ingeniero Superior de Telecomunicaciones presenta

Autor: **D. John Michael Fuhrer Soria**

TUTOR: **Prof. D. Miguel Ángel Aguirre Echánove**

I. MARCO CONCEPTUAL

MARCO CONCEPTUAL

INTRODUCCIÓN

Una FPGA es un sistema compuesto de celdas funcionales programables, que se conectan entre si y con el exterior de una forma a su vez programable. Las siglas FPGA corresponden a *Field Programmable Gate Array*, lo cual puede traducirse como "matriz de puertas programable sobre la marcha".

Dentro de esta definición cabe una gran diversidad de tecnologías, y cada fabricante del sector aporta una filosofía propia en sus desarrollos. Sin duda se trata de un sector que se encuentra en auge desde hace varios años, sobre todo debido a los grandes avances que se han ido produciendo. Hoy en día las FPGAs superan la velocidad de proceso de 100 MHz, y la capacidad de 200.000 puertas, con una tendencia que apunta aún mucho más alto en los próximos años.

Parte de culpa de todo esto la tiene la gran competitividad que reina en el sector, donde empresas como Xilinx, Altera, Actel y otras se disputan el mercado, copando las dos primeras a partes iguales el 80% del mismo. Estas compañías han conseguido que los diseños basados en FPGAs sean viables donde antes no lo eran, convirtiéndose en una opción válida y rentable cada vez más frente a los diseños a medida o *full custom*.

Pero para que un fabricante alcance el éxito con sus FPGAs no es suficiente con que el diseño del hardware sea acertado, sino que este debe estar acompañado por un paquete de herramientas CAD sencillo y completo para el usuario, y que además aproveche de forma óptima las capacidades que ofrece el hardware de la FPGA.

Esto último es lo que se ha pretendido con el estudio y los desarrollos software que son objeto de esta memoria para las tarjetas FPGA del proyecto FIPSOC, concretamente para la fase de mapeo tecnológico de la que hablaremos más adelante.

El diseño basado en FPGAs permite al usuario realizar por sí mismo todas las fases hasta la implementación, incluyendo simulación previa y pruebas. El único coste por hardware es el que se tiene que pagar por la adquisición de la FPGA y las herramientas de diseño.

Para acercarnos al objetivo de este proyecto, empezaremos viendo por encima las dos filosofías principales en la construcción de FPGAs, haciendo especial hincapié en el concepto de LUT (*Lookup Table)*. Luego veremos el flujo de diseño con FPGAs, para identificar la fase de mapeo tecnológico (*technology mapping*) en la cual está centrado este proyecto. Por último hablaremos del concepto de FPGAs multicontexto.

CÉLULAS DE GRANO FINO Y DE GRANO GRUESO

Como acabamos de decir, existen dos tendencias o filosofías diferentes en la construcción de FPGAs. Algunos fabricantes proponen diseños basados en células de grano fino, mientras que otros tienden al uso de células de grano grueso.

En las FPGAs de grano fino, las células pueden llegar a ser tan pequeñas que consistan tan solo en un transistor. Por el contrario, las celdas de grano grueso pueden contener por ejemplo un conjunto de multiplexores, o bien un elemento de memoria además de un grupo de biestables, formando unidades de gran tamaño y complejidad, pero también versatilidad.

Las FPGAs del proyecto FIPSOC contienen células de grano grueso. La estructura de estas células se verá en detalle más adelante, pero es preciso adelantar que dicha estructura se basa fundamentalmente en el uso de LUTs.

LUTs (LOOKUP TABLES)

Una LUT es un elemento capaz de producir cualquier función de salida a una serie de entradas. Hablaremos de K-LUTs para referirnos a LUTs de K entradas.

Para conseguir esto se suelen usar elementos de memoria que contienen almacenada la salida adecuada para cada combinación a la entrada. Así, una LUT de K entradas o K-LUT se constituirá mediante una RAM de 2^K posiciones y un bit de salida. Con ello obtenemos cualquier función de K entradas y una salida por compleja que sea, es decir, podemos implementar cualquier conjunto de puertas lógicas siempre que tenga K o menos entradas y una sola señal de salida.

Por ejemplo, para realizar mediante una 4-LUT la función siguiente:

aoi22: **O = !(a*b + c*d)**

usaríamos un elemento de memoria RAM con 4 entradas y 16 posiciones, con la información almacenada que se muestra en la tabla siguiente:

d	c	b	a	**O**
0	0	0	0	**1**
0	0	0	1	**1**
0	0	1	0	**1**
0	0	1	1	**0**
0	1	0	0	**1**
0	1	0	1	**1**
0	1	1	0	**1**
0	1	1	1	**0**
1	0	0	0	**1**
1	0	0	1	**1**
1	0	1	0	**1**
1	0	1	1	**0**
1	1	0	0	**0**
1	1	0	1	**0**
1	1	1	0	**0**
1	1	1	1	**0**

FLUJO DE DISEÑO EN UNA FPGA

El proceso de introducción del diseño de un circuito por parte de un usuario en una FPGA comprende una serie de pasos, desde la creación de dicho diseño mediante las herramientas CAD correspondientes hasta la grabación en la FPGA para el comienzo del uso.

Esto es lo que se conoce como flujo de diseño en la FPGA, y aunque puede variar según el modelo y el fabricante, en general se suele ajustar al diagrama que podemos observar en la figura I.1.

El Mapeado Tecnológico (*Technology Mapping*) es la primera fase del *Place & Route*, y se encarga de definir el número de celdas que se son necesarias para implementar un diseño en una FPGA y la configuración interna de cada una de ellas.

En la fase de *Partitioning*, cuando un circuito es demasiado grande para una FPGA, se divide este en dos o más partes para introducir cada parte en un dispositivo hasta completar la implementación.

A continuación, el *Placement* asigna cada bloque lógico identificado durante al mapeado a una celda de la FPGA.

Por último, el *Routing* realiza la asociación de las conexiones físicas en la FPGA a las conexiones lógicas.

El resultado final de la etapa de *Place & Route* debe ser un *Bit-Stream* (ristra de bits) que se conoce como mapa de programación, y que contiene toda la información necesaria para la grabación del diseño en la FPGA.

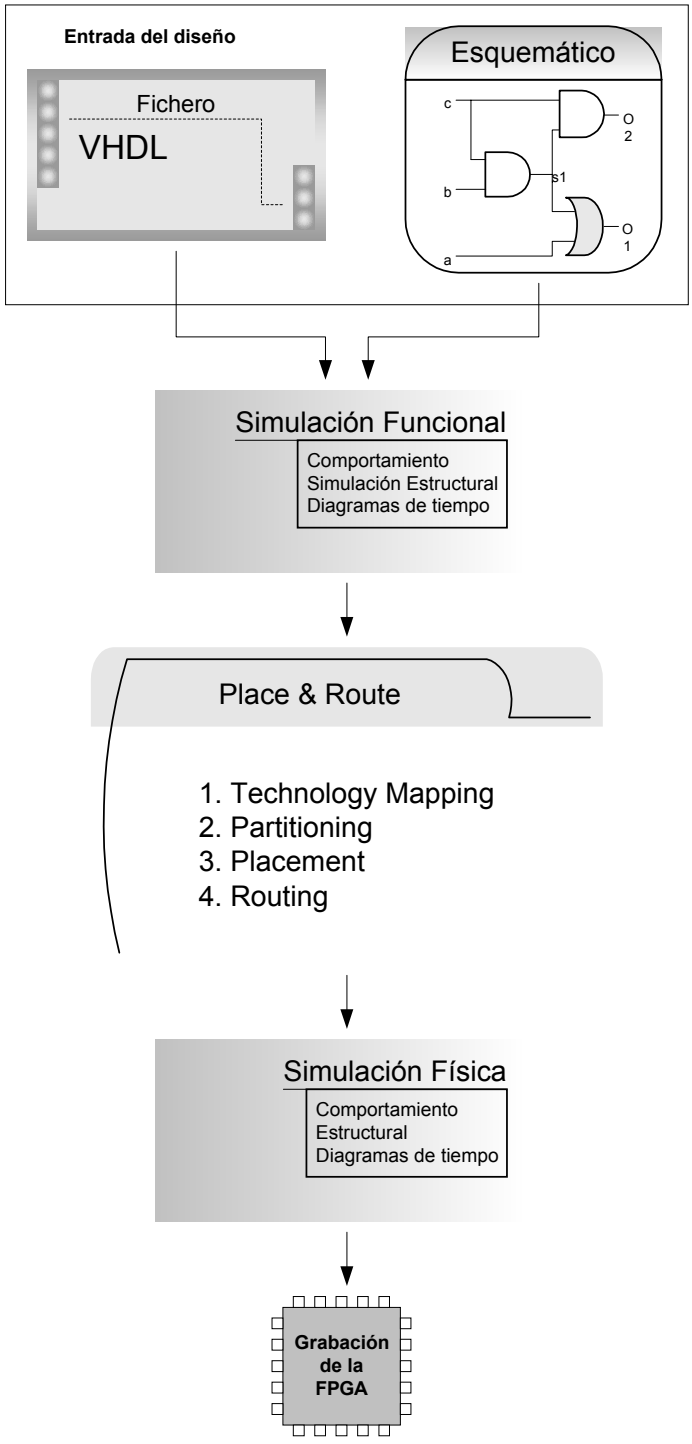


Fig. I.1: Flujo de diseño en una FPGA

TECHNOLOGY MAPPING

Este proyecto está orientado al desarrollo de un algoritmo eficiente de mapeo tecnológico orientado a la optimización de área. Las FPGAs que nos ocuparán están basadas en LUTs, y el objetivo final del algoritmo será el de minimizar el número de LUTs necesarias para la implementación de un circuito.

La tarea del desarrollo del algoritmo es de una gran complejidad, sobre todo porque comprende también la creación de los interfaces de entrada y de salida, mediante ficheros que deben ajustarse a un formato especificado. Por ello se ha considerado adecuado limitar a la realización de dicha tarea el ámbito de este proyecto fin de carrera.

Pero en la práctica, un algoritmo para *Technology Mapping* debiera estar además orientado no sólo a la optimización de área o de tiempo de retardo, sino también a la posterior *rutabilidad*.

En efecto, puede ocurrir que el mapeo óptimo en área de un circuito de lugar a un conglomerado de bloques lógicos cuyo ruteo sea de tal complejidad que haga que el diseño resultante no cumpla las características esperadas, o simplemente sea inviable su ruteo debido a los limitados recursos del dispositivo. Sin embargo, la adición del problema de la *rutabilidad* complica el problema por encima de las aspiraciones de este proyecto.

En cualquier caso, creo conveniente mencionar que se han desarrollado algoritmos en este sentido, como por ejemplo *Wmap*, diseñado para optimizar a la vez *rutabilidad* y eficiencia (en área y retardo), dando prioridad a la *rutabilidad*. El objetivo que se plantea en Wmap es el de minimizar en primer lugar el número total de *wires* o conexiones en el circuito, y después la eficiencia del circuito resultante.

Existen multitud de planteamientos para la resolución del problema del mapeo tecnológico, y hasta la fecha se han desarrollado muchos algoritmos diferentes también con diferentes objetivos de optimización. Existen desarrollos basados en sistemas expertos fundamentados en reglas y costes, otros basados en librerías de funciones y descomposición del circuito, etc.

En cuanto a algoritmos específicamente desarrollados para FPGAs basadas en LUTs cabe destacar Mis-pga, Asyl, Hydra, Xmap, VISMAP, DAG-Map y Chortle, en el cual se basa el desarrollo de este proyecto.

FPGAs DINÁMICAS MULTICONTEXTO

Las FPGAs tradicionalmente almacenan un único contexto *on-chip* para controlar el funcionamiento del dispositivo. Esta configuración da lugar a un sistema que es eficiente realizando su función a una alta velocidad de reloj. Pero en la práctica, la mayoría de los recursos de una FPGA se encuentran en estado de espera gran parte del tiempo que dura el ciclo de reloj. Para aprovechar más eficazmente los recursos de la FPGA para realizar diferentes operaciones se les pueden añadir múltiples contextos configurables dinámicamente.

El interés de estas modificaciones estriba en que el área que se requiere para cada configuración adicional es pequeña comparada con el área activa que controla. Mapas de programación para las FPGAs tradicionales con largos caminos críticos, bajos requerimientos de *throughput* o de lógica cambiante con el tiempo pueden ser automáticamente mapeados en implementaciones multicontexto que requieren del orden de la tercera parte de área que en un único contexto.

El ahorro de área puede resultar aún mayor cuando los diseños se realizan a propósito para la implementación multicontexto.

Como veremos en las siguientes secciones relativas a la estructura interna de las FPGAs del proyecto FIPSOC, estas son configurables en modos dinámicos para almacenar diferentes contextos de forma que se pueden poner activos alternadamente sin parar el funcionamiento del dispositivo.

1. FPGAs DEL PROYECTO FIPSOC

El circuito integrado del proyecto FIPSOC (Field Programmable System On Chip) contiene tres elementos principales:

- Un FPAA (Array Analógico Programable).
- Un núcleo microprocesador integrado que actúa como un elemento procesador de propósito general y además configura las células programables y sus interconexiones.
- El array de células programables, FPGA.

La FPGA está compuesta de un array de DMCs (Macro Celdas Digitales) rodeado por IOBs (Bloques In/Out) y por IICs (Células de Interfaz Interna) como se aprecia en la figura 1.1.

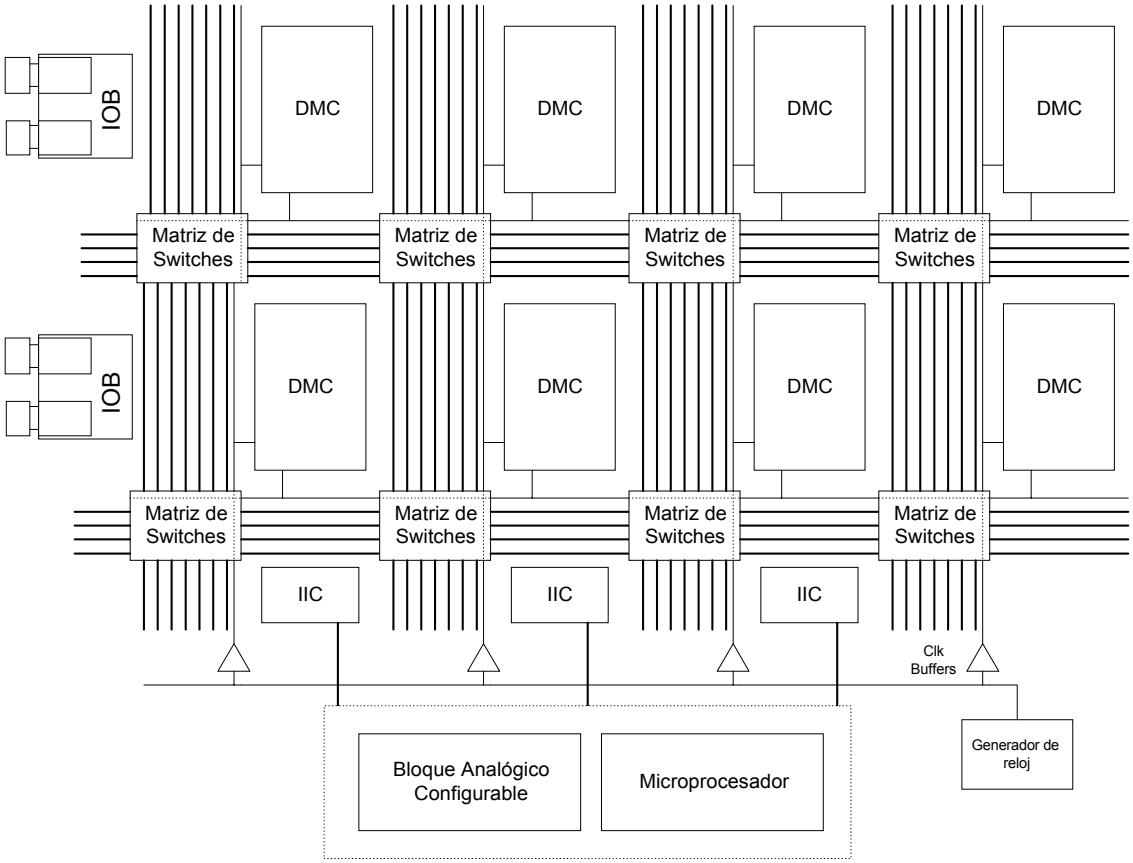


Fig. 1.1: Estructura de la FPGA del proyecto FIPSOC

1.1 LA MACRO-CÉLULA DIGITAL (DMC)

Proporciona dos formas de operación: estática y dinámica. En los modos dinámicos, se pueden almacenar dos contextos diferentes en bits RAM, de forma que es posible pasar de un contexto a otro mediante una simple operación de intercambio o *hardware swap* sin afectar al funcionamiento del dispositivo. Es posible reconfigurar el contexto no activo mientras el otro contexto está activo, sin perturbar su funcionamiento, lo que abre un enorme campo de posibilidades de aplicación para estos dispositivos.

Sin embargo, en este proyecto se hace uso de los modos estáticos de funcionamiento exclusivamente, ya que el objetivo es el de desarrollar un algoritmo eficaz para el mapeo de circuitos y no tanto el de aprovechar todas las posibilidades que ofrece la FPGA del proyecto FIPSOC.

En los puntos siguientes veremos con más detalle las posibles configuraciones del dispositivo en los modos estáticos de operación.

- **Diagrama de bloques de la DMC:**

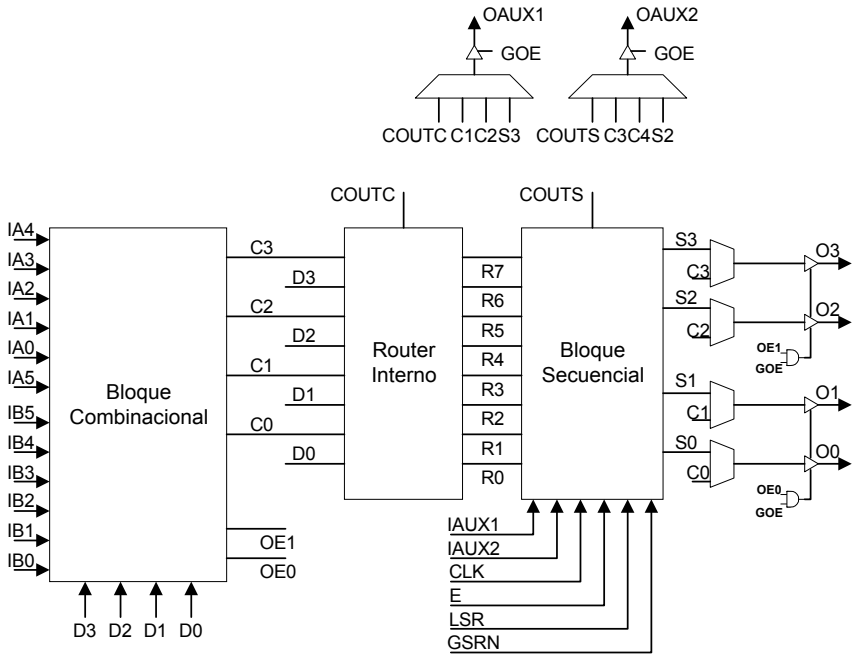


Fig. 1.2: Diagrama de bloques de la Macro Célula Digital (DMC)

- **Señales de E/S:**

Las FPGAs del proyecto FIPSOC son dispositivos complejos con muchas posibilidades de configuración. Vamos a describir el uso de las señales más relevantes tan sólo para aquellos modos de los que se hace uso en este software para technology mapping.

IA0-IA5, IB0-IB5: Son los pins de entrada a las LUTs, las cuales pueden ser, en los modos estáticos, de 4 o de 5 entradas.

D0-D3: Entadas de datos principalmente usadas como entrada directa al bloque secuencial. Proporcionan la posibilidad de utilizar las partes secuencial y combinacional del DMC de forma totalmente independiente.

CLK: Entrada de reloj. Su polaridad es configurable por flancos de subida o de bajada y por niveles alto o bajo. En este proyecto sólo utilizo *Flip-Flops* (FFs) como elementos secuenciales, los cuales requieren una entrada de reloj por flanco de subida o de bajada.

E: *Enable*, activo a nivel alto para los FFs tipo D con enable.

LSR: *Local Set/Reset* para los FFs, señal activa a nivel alto. Configurable como Set o como Reset para cada FF individualmente.

GSRN: *Set/Reset* Global Asíncrono para los FFs. Configurable como Set o como Reset para cada FF individualmente.

O0-O3: Señales de salida de la DMC.

- **Señales Internas:**

GOE: Esta señal de entrada está conectada al *Global Output Enable* y es una señal común en todas las DMCs y los IOBs de la FPGA. Se puede usar en el arranque para deshabilitar todos los pines de salida de todas las DMCs y los IOBs, evitando así colisiones por salidas arbitrarias.

C0-C3: Salidas del bloque combinacional generadas por las LUTs y su conectividad interna.

R0-R7: Salidas del router interno. Se corresponden sencillamente con las entradas C0-C3 y D0-D3 permutando el orden, aunque no todas las combinaciones son posibles.

S0-S3: Salidas del bloque secuencial, en el caso de este proyecto, de los FFs.

OE0, OE1: Señales *Output Enable*, que permiten dejar la salida en triestado mediante los *buffers* de salida.

- **Datos de Configuración:**

La configuración de cada DMC se guarda en bits RAM, que están duplicados para almacenar dos contextos independientes. Estos bits se pueden separar en tres grupos según la forma en la que se denominan:

- Los que comienzan con **CC:** configuran el bloque combinacional.
- Los que comienzan con **CR:** configuran el router interno.
- Los que comienzan con **CS:** configuran el bloque secuencial

Así por ejemplo, para los modos estáticos tendremos siempre **CCR** = 0. A continuación veremos las configuraciones de los modos que nos interesan en el ámbito de este proyecto.

1.1.1 Bloque Combinacional

Dentro del bloque combinacional se distinguen dos sub-bloques llamados *tiles*, y tenemos así el tile superior y el tile inferior. En función de los bits **CCMA0** y **CCMB0** se configura el tile inferior, y en función de **CCMA1** y **CCMB1** el superior. En los modos estáticos, cada tile se puede configurar independientemente como dos LUTs de 4 entradas o como una sola LUT de 5 entradas. También se pueden configurar como multiplexores 4 a 1 y como memorias RAM de 16x2, aunque estas son configuraciones que no se usan en el ámbito de este proyecto.

Tenemos entonces cuatro posibles formas de utilizar LUTs en el bloque combinacional, como vemos en las figuras 1.3 a 1.6.

CCMA0 = 0, CCMB0 = 0, CCMA1 = 0, CCMB1 = 0:

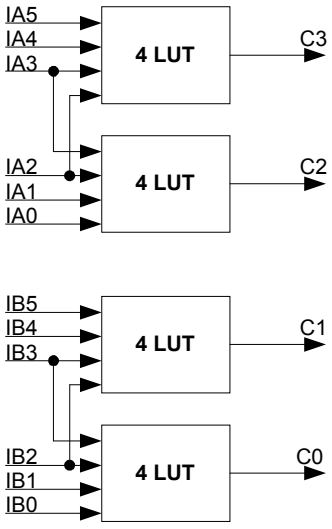


Fig. 1.3: Modo Combinacional Estático Simple

CCMA0 = 1, CCMB0 = 1, CCMA1 = 1, CCMB1 = 1:

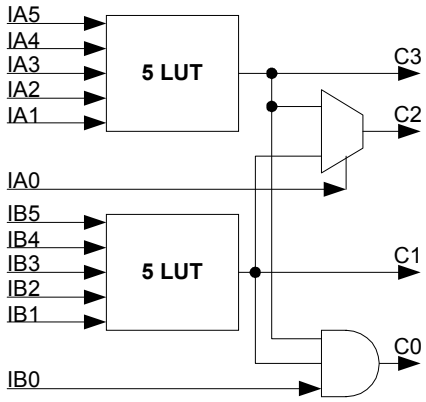


Fig. 1.4: Modo Combinacional Estático Complejo

CCMA0 = 0, CCMB0 = 0, CCMA1 = 1, CCMB1 = 1:

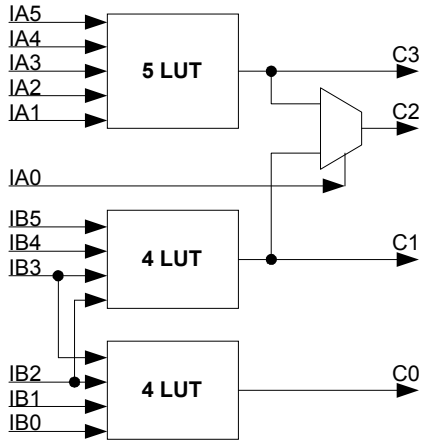


Fig. 1.5: Modo Combinacional Estático Complejo Superior-Simple Inferior

CCMA0 = 1, CCMB0 = 1, CCMA1 = 0, CCMB1 = 0:

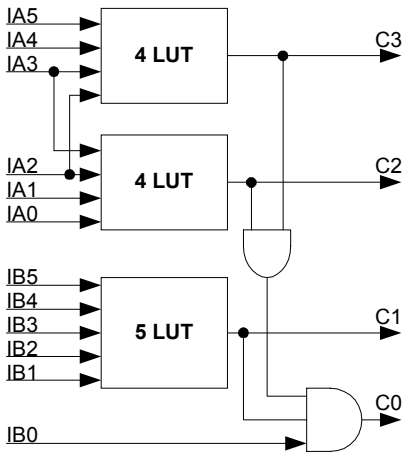


Fig. 1.6: Modo Combinacional Estático Simple Superior-Complejo Inferior

En los modos de operación dinámica el bloque combinacional puede albergar LUTs de 3 y de 4 entradas, así como dos multiplexores 4 a 1 o dos memorias RAM 8x2. También existe un modo especial en el que el dispositivo se configura como un sumador de 4 bits.

No obstante, intentar aprovechar de forma óptima todas estas posibilidades aumentaría la complejidad del software en exceso para el ámbito de este proyecto. El uso de los modos dinámicos para almacenar dos contextos puede permitir la implementación de circuitos de mayor área, que sobrepasen la capacidad del dispositivo en modos estáticos. Pero el problema de la división de un circuito en dos partes, correspondientes a dos contextos, y la configuración de estos en la FPGA más el *hardware swap* es de tal complejidad que requiere un amplio estudio y podría ser el tema de investigación para otro proyecto completo.

Por eso me he limitado a utilizar las configuraciones que hemos visto en las figuras anteriores, correspondientes a los cuatro diferentes modos estáticos de operación con LUTs.

1.1.2 Bloque Secuencial

Este bloque está diseñado para ofrecer una interfaz flexible para la parte combinacional. Tiene un ancho de 4 bits, y soporta un gran número de diferentes tipos de Flip-Flops que pueden ser usados de forma más o menos independiente.

El esquema general del bloque secuencial es el que se puede apreciar en la figura 1.7.

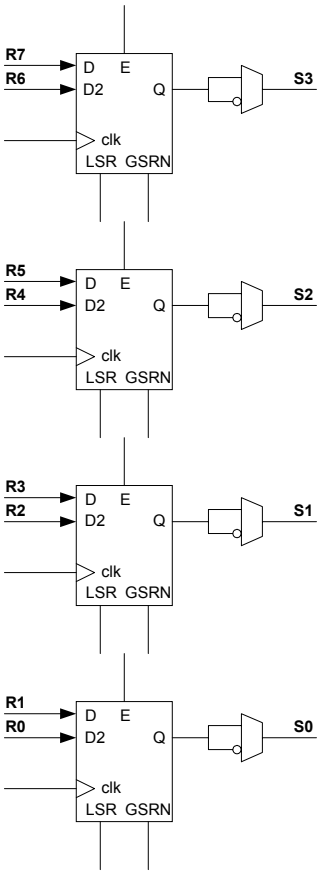


Fig. 1.7: Estructura Interna del Bloque Secuencial

La configuración del bloque secuencial se almacena en los bits CSR y CS1 a CS9. No sólo esta información está duplicada para que sea posible operar con dos contextos independientes, sino también la información almacenada por los FFs en cada contexto.

Mediante el bit CSR podemos seleccionar si la información de los FFs es distinta para cada contexto o es compartida por ambos contextos.

Con los bits CS1 a CS9 podemos configurar el dispositivo para obtener distintos tipos de FFs más o menos independientes. O bien utilizando los macro-modos de 4 bits se consiguen un registro de desplazamiento o un contador en lugar de los cuatro FFs.

En el ámbito de este proyecto me limito al uso de FFs. En el fichero de librería (ver anexo 1) se encuentran recogidos distintos FFs con la correspondiente información necesaria para la configuración.

En las figuras 1.8 a 1.11 vemos los cuatro tipos de FFs que se pueden seleccionar en función de los bits CS1 a CS3.

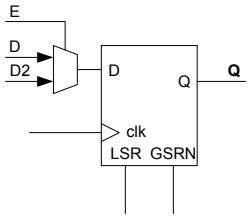


Fig. 1.8: FF Tipo Multiplexor

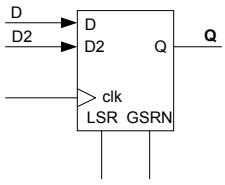


Fig. 1.9: FF Tipo D con Reset Local Síncrono (LSR)

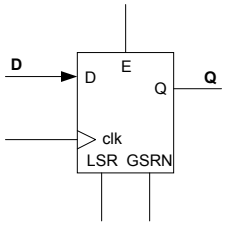


Fig. 1.10: FF Tipo D con Enable

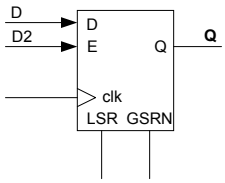


Fig. 1.11: FF Tipo D con Enable Local

1.1.3 Router Interno

Todos los recursos de ruteo de la DMC, tanto el Router Interno como la conectividad a la salida (que proporciona versatilidad en la interconexión de DMCs), son reconfigurables dinámicamente. Por eso también aquí están duplicados todos los bits de configuración para ser intercambiados cuando el microprocesador lanza el comando de *hardware swap*.

El Router Interno proporciona conectividad entre los bloques combinacional y secuencial. En la figura 1.12 vemos su estructura interna.

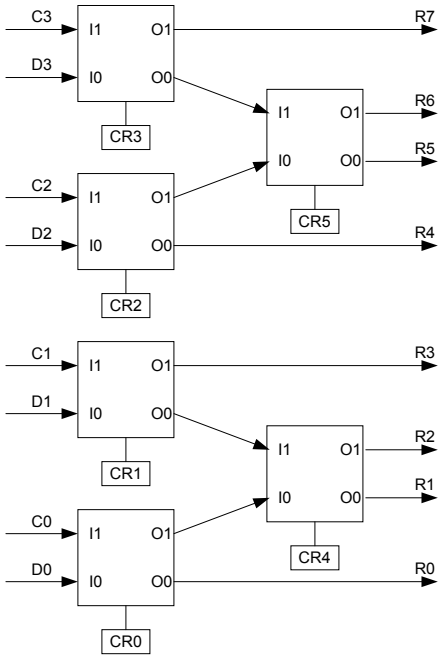


Fig. 1.12: Estructura Interna del Router de la DMC

Mediante los bits CR0 - CR5 se controlan los seis selectores de entrada que contiene el router. Para cada uno de ellos, si el bit de selección es 1 entonces O1 se cortocircuita a I1 y O0 a I0, y si por el contrario el bit es 0 se conecta O1 a I0 y O0 a I1.

Por otra parte, la DMC también ofrece control sobre la conectividad de salida. Los bits de control CR6 - CR9 y CRO3 - CRO0 son los que realizan la selección en los multiplexores de salida del dispositivo que vimos en la figura 1.2, de donde se obtienen las señales de salida O0 - O3, OAUX1 y OAUX2.

1.2 INPUT-OUTPUT BLOCKS (IOBs)

Los IOBs o Bloques de Entrada-Salida son las células que permitan la conexión de la FPGA con el exterior. Se disponen en la periferia del dispositivo, como vimos en la figura 1.1.

Son configurables como entrada, salida o señales bidireccionales, y pueden programarse como señales directas o negadas. La capacidad de dar señal o de *fan-out* de los buffers de salida es también programable.

1.3 RECURSOS DE RUTEO

Como se vio en el esquema general de la figura 1.1, la conexión entre las DMCs de la FPGA se realiza mediante canales que la recorren vertical y horizontalmente, los cuales se cortan en las Matrices de Switches.

1.3.1 Canales de conexión

Hay 24 canales verticales por columna y 16 horizontales por fila, sin incluir los canales especiales para la señal de reloj. No todos los canales son iguales, tienen distinta longitud y están destinados a distintos tipos conexiones, desde locales a conexiones entre bloques distantes o a señales globales como resets o enables.

La arquitectura de ruteo está programada también mediante bits de configuración que se ubican en el mapa de memoria dentro de algún bloque dado de la FPGA, como una DMC o un IOB.

1.3.2 Matrices de Switches

Las Matrices de Switches se usan para interconectar los canales verticales con los horizontales, y están implementadas mediante multiplexores bidireccionales. Por ejemplo, para un canal horizontal tendremos siete switches o interruptores para conectarlo a canales verticales, pero sólo uno podrá estar activo.

La configuración también viene dada por bits de control que se guardan en el mapa de memoria correspondiente al bloque más cercano que tenga la matriz de switches hacia la parte superior derecha.

2. TECHNOLOGY MAPPING. FORMATOS DE LOS FICHEROS.

Una vez que hemos visto la arquitectura de las FPGAs del proyecto FIPSOC ya podemos abordar el problema de Technology Mapping.

La DMC o Macro-Celda Digital tiene tres bloques secuenciales como se vio en el apartado anterior: el bloque combinacional, el secuencial y un router interno que permite flexibilidad en las conexiones entre ambos.

El primer paso a la hora de abordar el mapeo de un circuito será el de separar las partes combinacionales (puertas lógicas) de las secuenciales. En este proyecto, como ya he indicado anteriormente, sólo tendremos en cuenta flip-flops como elementos secuenciales. A partir de entonces se puede dar por concluido el problema para la parte secuencial, simplemente quedará enumerar los flip-flops con sus correspondientes señales de entrada y salida.

Para las partes combinacionales, el problema de Technology Mapping se reduce a la división de un circuito en pequeños bloques, cada uno de los cuales debe tener cabida en una LUT de tamaño K. Estos bloques serán conjuntos de puertas lógicas NOT, AND, OR y XOR, agrupadas de forma que sumen un máximo de K entradas y una sola salida.

Para resolver el problema, en este proyecto se lleva a la práctica un algoritmo de mapeo cuyo objetivo es el de minimizar el número de LUTs utilizadas. El resultado final es un programa que recibe un fichero de entrada con la descripción de un circuito, y tras el procesado produce un fichero de salida para la siguiente fase en el flujo de diseño de la FPGA.

El programa requiere también la presencia de un fichero de librería donde se describen los distintos tipos de puertas lógicas y FFs admisibles. A continuación veremos los formatos de estos ficheros.

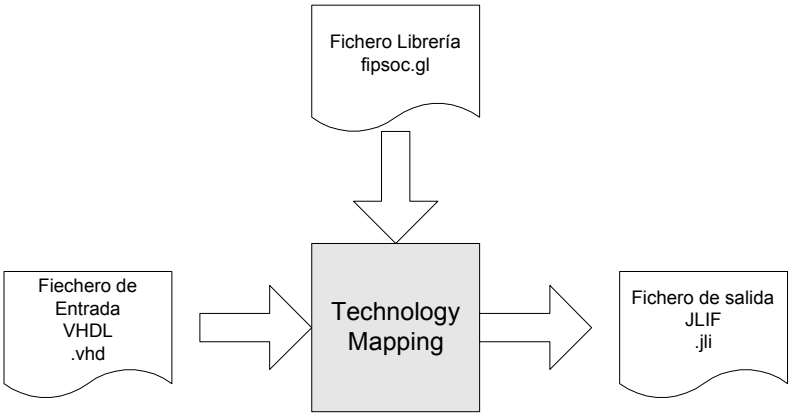


Fig. 2.1: Ficheros relacionados

2.1 VHDL

Las siglas VHDL provienen de VHSIC (*Very High-Speed Integrated Circuits*) *High Design Lenguaje*. Este lenguaje permite la completa descripción de un circuito, y es el formato en que vendrá dado el fichero de entrada al programa de mapeo.

El fichero VHDL se produce en una fase previa, en la cual el usuario introduce el diseño del circuito en el sistema, ya sea directamente escribiendo código VHDL o bien de una forma gráfica. Los modernos entornos de diseño para FPGAs ofrecen muchas facilidades para ello. Entonces, los datos introducidos por el usuario son chequeados y tras corregirse los posibles errores de usuario se obtiene el fichero VHDL que será nuestro fichero de entrada.

En este proyecto dicho fichero de entrada debe venir dado como un fichero de texto plano. En las figuras 2.2 y 2.3 vemos un pequeño circuito de ejemplo y su correspondiente descripción mediante VHDL.

El circuito contiene puertas lógicas y un flip-flop tipo D sin reset ni enable. Hay un total de 7 entradas o *in-ports* (i_1-i_7) y una salida o *out-port* (o_1).

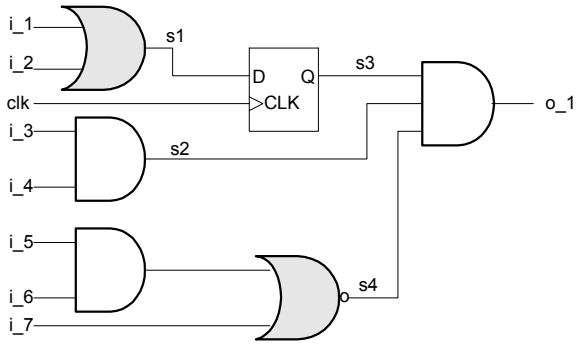


Fig. 2.2: Circuito de entrada

```
library IEEE;
use IEEE.std_logic_1164.all;
entity top is
    port ( i_1, i_2, i_3, i_4, i_5, i_6, i_7, clk: in std_logic;
           o_1: out std_logic);
end top;

architecture SYN_blif of top is
        component or2
          port ( a, b: in std_logic; O: out std_logic);
        end component;

        component and2
          port ( a, b: in std_logic; O: out std_logic);
        end component;

        component and3
          port ( a, b, c: in std_logic; O: out std_logic);
        end component;

        component aoi21
          port ( a, b, c: in std_logic; O: out std_logic);
        end component;

        component DFF
          port (D, CLK: in std_logic; Q: out std_logic);
        end component;

signal s1, s2, s3, s4: std_logic;

begin
U1: or2 port map ( a => i_1, b => i_2, O => s1);
U2: and2 port map ( a => i_3, b => i_4, O => s2);
U3: and3 port map (a => s3, b => s2, c => s4, O => o_1);
U4: aoi21 port map ( a => i_5, b => i_6, c => i_7, O => s4);
FF1: DFF port map (D => s1, CLK => clk, Q => s3);

end SYN_blif;
```

Fig. 2.3: VHDL correspondiente al circuito de la fig. 2.2

El código VHDL de la figura 2.2 es un ejemplo característico en el que se distinguen varios bloques tras las dos primeras líneas (que aparecerán siempre):

- Declaración de la entidad principal, "*top*" en este caso, y enumeración de las señales de entrada y salida. La señal de reloj *clk* es una entrada más.

- Declaración de los distintos componentes que se usan en el circuito: puertas simples (AND, OR, XOR), elementos combinacionales complejos o grupos de puertas (aoi21, aoi22, mux21, ...) y flip-flops. En este proyecto no se admiten otros elementos secuenciales como registros de desplazamiento.
  Cada uno de estos componentes debe tener su definición correspondiente en el fichero de librería.

- Enumeración de las señales internas que conectan entre sí los elementos del circuito.

- Descripción completa del circuito enumerando cada componente con los nombres de sus señales a la entrada y a la salida.

Nótese en este ejemplo la inclusión de un elemento complejo como es el denominado *aoi21*, que en el listado VHDL constituye U4. Si atendemos al fichero de librería que aparece en el anexo 1, podemos comprobar que la definición del aoi21 se corresponde con el conjunto de dos puertas, una AND-2 y otra NOR-2, dispuestas tal como aparecen en el circuito de la figura 2.2 y donde las entradas son i_5, i_6 e i_7 y la salida es s4.

El resto de puertas y el flip-flop DFF también se encuentran definidos en el fichero librería, entre muchos otros elementos que no se han usado en este ejemplo.

2.2 EL FICHERO DE LIBRERÍA "fipsoc.gl"

Como veremos más adelante en esta memoria, la primera fase que ejecuta el programa de mapeo es la lectura del fichero de entrada VHDL. En esta fase es cuando el programa requiere abrir también el fichero librería con las definiciones de los componentes.

El anexo 1 recoge el listado completo del fichero librería. Se divide en dos bloques, el primero de los cuales corresponde a las definiciones de puertas y elementos combinacionales más complejos. A partir de la línea

FLIP_FLOPS;

comienzan las definiciones de los elementos secuenciales, en este caso FFs.

El símbolo almohadilla (#) y el doble signo menos (- -) en una línea indican que desde ese punto hacia la derecha y hasta comenzar una nueva línea todo serán comentarios, es decir, será ignorado por el programa.

2.2.1 Definiciones de elementos combinacionales

Cada elemento debe definirse empezando por la palabra "GATE" seguida del nombre que se le quiera dar a ese elemento.

A continuación y con una separación de al menos un espacio vendrá el número de pines de entrada (el número de pines de salida no es necesario puesto que siempre será 1).

Por último debe aparecer la definición de la función lógica que realiza el elemento combinacional expresada mediante las operaciones lógicas fundamentales, cuyos símbolos son los convencionales:

NOT	!
OR	+

AND *
XOR ^

Se permite también el uso de paréntesis para elevar la complejidad del elemento de forma ilimitada. Algunos ejemplos son:

```
GATE or2        2    O=a+b;
GATE aoi22      4    O=!(a*b+c*d);
GATE mux21      3    O=!(!a*s+!b*!s);
```

Es obligatoria la terminación de la línea con punto y coma. Nótese que las definiciones que aparecen en el fichero son interpretadas por el programa según se va leyendo el código VHDL de entrada. Esto le proporciona gran funcionalidad, puesto que el fichero de librería se puede editar por el usuario para incorporar nuevos elementos o modificar los que ya aparecen si fuera conveniente.

Las modificaciones en este fichero tienen por tanto efectos inmediatos en el funcionamiento del programa. Véase esto por ejemplo en el hecho de que existen tres formas alternativas para la definición de la puerta XOR:

```
GATE xor2       2    O=a^b;

# GATE xor2     2    O=a*!b+!a*b;    -- Formas alternativas

# GATE xor2     2    O=!(a*b+!a*!b);
```

Las dos segundas definiciones están comentadas, aunque de no ser así el efecto sería el mismo puesto que cuando el programa busca en el fichero se queda con la primera definición que encuentra.

En este caso, una puerta XOR se trata como tal, es decir, como una sola puerta. Pero si se comentase la primera definición en lugar de la segunda o la tercera, cada puerta XOR que incluyese el fichero de entrada VHDL se sustituiría en la práctica por un conjunto de puertas AND, OR y NOT según la definición equivalente elegida.

En principio es preferible la primera opción, puesto que lo más probable es que utilizando una sola puerta XOR en lugar de varias AND y OR más inversores el área ocupada total sea menor, y de hecho así lo confirman los resultados en la práctica.

2.2.2 Definiciones de elementos secuenciales

Los flip-flops (FFs) se definen de la siguiente forma: en primer lugar se debe indicar el nombre del tipo de FF que se trate.

A continuación se deben enumerar los nombres de las señales de entrada y salida, indicando "NC" cuando una de ellas quede sin conectar.

Por último debe aparecer una ristra de diez bits terminada en punto coma. Como veremos en el siguiente punto, estos bits definen la configuración del FF en el bloque secuencial de la Macro-Celda Digital (DMC) (ver figura 3.2) según el formato JLIF.

Por ejemplo, un FF tipo D sin enable ni reset se define como sigue:

```
DFF          D NC CLK VDD GND VDD Q          1000010000;
```

En el fichero librería aparecen las definiciones de los diferentes FFs que se han usado en las pruebas del programa, los que aparecen en los ficheros VHDL de prueba.

2.3 FORMATO JLIF

JLIF es el formato de entrada para el programa *fitter*, que lleva a cabo la fase siguiente al technology mapping en el flujo de diseño de la FPGA. El fitter recoge la información en forma de fichero JLIF necesaria para la configuración de todas las

macro-celdas o DMCs necesarias para un diseño, en nuestro caso serán LUTs, *pads* de entrada y salida, y flip-flops.

Para el fichero de entrada de la figura 2.3, por ejemplo, el programa da el fichero de salida de la figura siguiente.

```
.PAD p_i_1 NC i_1 GND 0 U -1 010000000100
.PAD p_i_2 NC i_2 GND 0 U -1 010000000100
.PAD p_i_3 NC i_3 GND 0 U -1 010000000100
.PAD p_i_4 NC i_4 GND 0 U -1 010000000100
.PAD p_i_5 NC i_5 GND 0 U -1 010000000100
.PAD p_i_6 NC i_6 GND 0 U -1 010000000100
.PAD p_i_7 NC i_7 GND 0 U -1 010000000100
.PAD p_clk NC clk GND 0 U -1 010000000100
.LUT lut1 i_1 i_2 NC NC s1 000 0111011101110111
.LUT lut2 i_5 i_6 i_7 s3 s2_o_1 000 0000000011100000
.LUT lut3 s2_o_1 i_3 i_4 NC o_1 000 0000000100000001
.FF FF1 s1 NC clk VDD GND VDD s3 1000010000
.PAD p_o_1 o_1 NC VDD 0 U -1 110000000000
```

Fig. 2.3: Fichero JLIF de salida para el circuito de la fig. 2.2

Examinando el fichero de salida vemos que tras el procesamiento con el algoritmo el programa ofrece una solución de mapeo utilizando tres 4-LUTs y un FF. El resto de líneas del fichero JLIF enumeran los PADs de E/S, los cuales siempre se configuran de la misma forma para cualquier circuito.

2.3.1 Formato JLIF para las LUTs de 4 entradas (4-LUTs)

Para las LUTs como se puede ver, la línea comienza con la palabra ".LUT" seguida del nombre que se le quiera dar a la LUT, que será siempre "lutN" donde N se va incrementando conforme se vayan necesitando LUTs.

A continuación se escriben las señales de entrada, indicando "NC" cuando no se haga uso de algunas, y luego el nombre de la señal de salida. El programa puede necesitar crear nuevas señales de salida de LUTs durante el proceso; entonces les asigna nombres automáticamente, como es el caso de " s2_o_1" en el ejemplo.

Por último viene la información binaria para la configuración de la LUT. Hay un primer grupo de tres bits que siempre serán todos 0 en este proyecto, ya que con eso se configura el dispositivo en modo estático. A continuación vienen los 16 bits correspondientes a la función de salida de la LUT de 4 entradas.

2.3.2 Formato JLIF para las LUTs de 5 entradas (5-LUTs o Combinational Tiles)

Aunque en el ejemplo anterior no se hizo uso de 5-LUTs, la forma de especificarlas en JLIF es análoga. Veamos un ejemplo:

```
.COMB_TILE L1 NC i_1 i_2 i_3 i_4 i_5 NC NC NC 0_o1 000110 00010001000100010001000101010101
```

En lugar de la palabra ".LUT" debe escribirse ".COMB_TILE" ya que el uso de una LUT de 5 entradas requiere el uso de un *tile* completo del bloque combinacional de la DMC.

A continuación aparece el nombre de la LUT, en este caso "L1", y los nombres de las cinco señales de entrada.

Lo que sigue son los nombres de las señales que entran en D0 y D1 y las que salen por C0 y C1 (ver figura 3.2). En este proyecto nunca usaremos D0, D1 ni C0, y la salida de la 5-LUT irá por C1, con lo cual se escribe "NC NC NC O_o1" en el fichero JLIF, donde O_o1 representa el nombre de la señal de salida de la LUT.

Lo siguiente ya es la información de configuración: los 5 primeros bits son una constante, "000110", para indicar modo estático y uso de 5-LUT en este *tile*. Por último vienen los 32 bits que constituyen la función de salida de la 5-LUT.

2.3.3 Formato JLIF para los Flip-Flops

Observando la estructura de la figura 3.7, para cada FF la línea JLIF tiene la siguiente forma:

.FF nombre D0 D1 clk E LSR GSRN Q <bits de configuración>

donde los bits de configuración son 10 y corresponden a CS2-CS9 (ver punto 3.2) más dos bits que serán siempre 0, indicando modo estático.

II. JUSTIFICACIÓN Y OBJETIVOS

JUSTIFICACIÓN Y OBJETIVOS

Este proyecto fin de carrera abarca una fase importante y compleja dentro del flujo de diseño de una FPGA como es el *Technology Mapping*. Pero no es un trabajo aislado, sino que se engloba dentro de un marco más amplio como es el proyecto FIPSOC.

El proyecto FIPSOC está desarrollado conjuntamente por una serie de empresas y universidades, y comprende el diseño de una FPGA completo más el desarrollo de las herramientas CAD de diseño.

La realización en sí del *Technology Mapping* está fundamentada en una amplia base teórica, ya que las FPGA son dispositivos que viven desde hace unos años un intenso desarrollo y que han evolucionado considerablemente ganando mucho terreno en el sector del diseño de circuitos integrados, con lo cual la literatura existente es bastante extensa.

Para la realización del núcleo del desarrollo de este proyecto, que es el algoritmo de mapeo tecnológico, tomaré como documento fundamental la tesis de Robert J. Francis, de la universidad de Toronto, titulada *Technology Mapping for Lookup-Table Based Field-Programmable Gate Arrays*.

Dicha tesis es el resultado de un gran trabajo de investigación, y revela una serie de metodologías y mecanismos de mapeo tecnológico para FPGAs basadas en LUTs, como las del proyecto FIPSOC, con distintos objetivos como son la optimización de area y de tiempo de retardo global en los diseños.

El objetivo de este proyecto fin de carrera es la interpretación de esa base teórica y su adaptación al caso del proyecto FIPSOC para desarrollar un software de Technology Mapping orientado a la optimización de ocupación de área para su incorporación al conjunto de herramientas CAD.

Una vez desarrollado el armazón del programa de mapeo, el objetivo es que sea competitivo frente a otros programas ya existentes. Esto se comprueba mediante

el testeo con un conjunto de ficheros benchmarks y la comparación de los resultados. Entonces debemos incluir como objetivo la investigación necesaria para introducir mejoras en el algoritmo hasta alcanzar los resultados deseados en las pruebas.

El programa debe ofrecer diferentes opciones como el trabajo con LUTs de 4 o de 5 entradas, así como incluir un fichero de librería de componentes configurable y modificable por el usuario.

Al estar destinado a su inclusión dentro del flujo de diseño de la FPGA, el programa debe ajustarse a los formatos, que ya estarán especificados, de los ficheros de entrada y de salida, en este caso VHDL y JLIF.

El software debe además eficiente y robusto, y tener una programación estructurada y dinámica haciendo un uso eficiente de los recursos del sistema en el que se instale.

Hacia el final de esta memoria se incluirá un apartado dedicado a los resultados de la fase de pruebas, donde podremos evaluar si todos estos objetivos se han cumplido o no y en qué medida.

III. METODOLOGÍA

METODOLOGÍA

En los apartados anteriores hemos establecido el punto de partida para la realización de este proyecto. Es ahora cuando comienza la labor de investigación y desarrollo necesaria para la consecución del mismo.

En los apartados correspondientes al marco conceptual hemos visto en detalle las características y la estructura interna de las FPGAs del proyecto FIPSOC que nos ocupan, así como los formatos de los ficheros asociados.

Luego hemos establecido los objetivos a conseguir mediante el desarrollo de este proyecto, y las condiciones que debe cumplir el programa resultante.

Tenemos ya por tanto una base sobre la que comenzar a construir todo el proyecto, lo cual haremos en una serie de fases. Es necesario dividir el trabajo en diferentes tareas que se irán cumplimentando sucesivamente y que son las siguientes:

1. **PLANTEAMIENTO DEL ALGORITMO DE MAPEO:** La primera tarea que se debe realizar es la de concretar el algoritmo que se va a aplicar a los circuitos de entrada. Esto requiere una labor de investigación y un estudio teórico que debe dar como resultado una serie de conclusiones en forma de seudocódigo. Dicho seudocódigo servirá de guía para la implementación del algoritmo, y de él dependerá completamente el grado de éxito en los resultados finales en cuanto a optimización de área.

2. **DEFINICIÓN DE VARIABLES Y ESTRUCTURAS DE DATOS:** Antes de comenzar la escritura del código es fundamental definir cómo se van a almacenar los datos necesarios en el programa. Esta es una tarea crítica, ya que una definición acertada de las estructuras de datos puede simplificar mucho el desarrollo, y por el contrario una definición desacertada puede hacer perder mucho tiempo y esfuerzo.

3. **DESARROLLO DE LA INTERFAZ DE ENTRADA:** El objetivo de esta tarea es el de conseguir que el programa abra el fichero VHDL de entrada y sea capaz de interpretar y almacenar en memoria el circuito que describe. Mediante reserva dinámica de memoria se irán almacenando todas las puertas, biestables y señales de conexión hasta conformar una imagen del circuito en memoria utilizando las estructuras definidas en la tarea anterior.

4. **IMPLEMENTACIÓN DEL ALGORITMO DE MAPEO:** Una vez que el programa es capaz de leer un fichero VHDL de entrada y almacenar el circuito en memoria podemos construir el algoritmo. Este procesará las estructuras de datos del circuito modificándolas, creando otras nuevas y eliminando las de cálculos intermedios hasta obtener un circuito resultante dividido en LUTs.

5. **DESARROLLO DE LA INTERFAZ DE SALIDA:** Una vez que se ha procesado el circuito con el algoritmo de mapeo y se ha obtenido un circuito resultante, el programa debe generar un fichero de salida en el formato correspondiente a partir de la información almacenada en memoria.

6. **FASE DE PRUEBAS:** Tras obtener una primera versión operativa del programa, se ha de evaluar la efectividad del algoritmo implementado sobre un banco de circuitos *benchmarks*. La comparación de los resultados con los de otros programas nos puede dar pistas para la mejora del algoritmo. Con la introducción de mejoras en el algoritmo de mapeo iremos generando sucesivas versiones del programa hasta que se alcancen los objetivos deseados.

En las secciones siguientes iremos viendo en detalle la forma de acometer cada una de estas tareas y los resultados obtenidos hasta la finalización del proyecto, momento en el cual evaluaremos el grado de éxito alcanzado en los objetivos prefijados.

IV. ANÁLISIS Y DESARROLLO DE LA HERRAMIENTA DE MAPEO LÓGICO

DESARROLLO DE LA HERRAMIENTA DE MAPEO LÓGICO

1. DESCRIPCIÓN DEL ALGORITMO

Un circuito tiene elementos combinacionales y elementos secuenciales conectados entre sí por múltiples líneas. Dado que el circuito de entrada vendrá descrito en lenguaje VHDL estructural, habremos de considerar como elementos combinacionales puertas de cualquier tipo, y como elementos secuenciales tendremos únicamente FLIPFLOPs.

Todos los tipos de puertas y FLIPFLOPs están recogidos en el fichero de librería que acompaña al programa, donde se indica el número de entradas de cada uno y se describe la función que realizan.

El objetivo de este algoritmo es el de minimizar el número de LUTs que serán necesarias para codificar las partes combinacionales del circuito de entrada. El algoritmo actúa de forma genérica frente al número de entradas de las LUTs que utilizaremos. Este número se conoce como parámetro K, y hablaremos de K-LUTs para referirnos a LUTs de K entradas.

Es condición necesaria para que sea posible aplicar el algoritmo que no haya puertas en el circuito con más de K entradas. En caso contrario habría que aplicar previamente otro algoritmo para dividir esas puertas en conjuntos de puertas más pequeñas. Esta condición se cumplirá siempre para K=4 y K=5, los valores que se utilizan en la práctica, ya que las puertas del circuito tendrán un máximo de cuatro entradas.

La estructura general del algoritmo se divide en una serie de fases consecutivas:

1) Separación de partes combinacionales y secuenciales.
2) Simplificación de los inversores.
3) Reducción del circuito a un conjunto de árboles de puertas.
4) Algoritmo de árbol.
5) Optimización de área por reconvergencia.
6) Optimización de área por replicación.

En los siguientes apartados vamos a ir viendo con detalle cada uno de estos puntos, utilizando ejemplos siempre que sea posible.

1.1 SEPARACIÓN DE PARTES COMBINACIONALES Y SECUENCIALES.

Los elementos secuenciales de un circuito son sencillamente cortados y aislados para dejar sólo las partes combinacionales. Cuando *cortamos* un elemento secuencial, sus señales de entrada pasan a ser salidas globales del circuito combinacional, y sus salidas serán entradas globales al circuito combinacional.

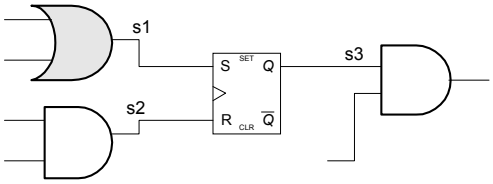


Fig. 1.1: Ejemplo de circuito con FF

En este ejemplo s1 y s2 se convertirían en salidas globales, y s3 en una entrada global. En la fase de Technology Mapping no nos ocupamos de la ubicación de los elementos secuenciales. El programa simplemente los aísla y los incluye en el fichero de salida con formato JLIF como ya hemos visto en el apartado 2.3.3.

Tras repetir esta operación con todos los elementos secuenciales del circuito nos quedará un circuito completamente combinacional conexo o inconexo en el que nos centramos para las siguientes fases.

1.2 SIMPLIFICACIÓN DE LOS INVERSORES.

Este es un paso necesario para hacer más sencillo el funcionamiento del algoritmo. Todos los inversores que aparecen en el circuito se eliminan de una forma u otra según su situación. Tomemos como ejemplo el siguiente circuito con 3 inversores:

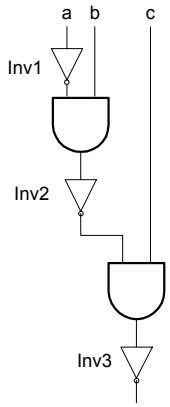


Fig. 1.2: Circuito con inversores

Siempre que un inversor se encuentre a la salida de otra puerta se eliminará transformando esa puerta en su inversa, es decir, si es una AND en una NAND, si es una OR en una NOR, etc.. Este es el caso de los inversores Inv2 e Inv3 de la figura.

Cuando el inversor no está a la salida de una puerta, como el Inv1 de la figura, se elimina marcando esa señal como negada a la entrada de todas las puertas a las que vaya.

Aplicando estas reglas obtenemos lo siguiente:

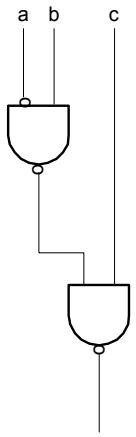


Fig. 1.3: Circuito tras suprimir los inversores

Tras finalizar este paso se obtendrá un circuito sin inversores para la siguiente fase del algoritmo. Esto supone una notable simplificación para los procedimientos subsiguientes, sobre todo desde el punto de vista de la implementación en código.

1.3 REDUCCIÓN DEL CIRCUITO A UN CONJUNTO DE ÁRBOLES DE PUERTAS.

Todo circuito se puede reducir a un conjunto de árboles de puertas simplificando los bucles que aparezcan en este. En este punto el programa realiza una búsqueda sobre todas las señales del circuito que pueden ser:

- Señales de entrada global del circuito (*IN ports*).
- Señales de salida global del circuito (*OUT ports*).
- Señales internas.

Esta distinción viene ya especificada en el fichero VHDL de entrada. El objetivo de la búsqueda es el de establecer cuáles de las puertas existentes van a ser los nodos raíz de los árboles de puertas. Tenemos una serie de reglas:

- Si una señal *IN port* aparece como salida de alguna puerta (el circuito se realimenta), esa puerta será raíz de un árbol.

- Si una señal *OUT port* aparece como salida de alguna puerta, esa puerta será también raíz de un árbol.

- Si una señal interna S es entrada en más de una puerta del circuito, la puerta de la cual S es salida será raíz de un árbol.

Tomemos el siguiente circuito como ejemplo:

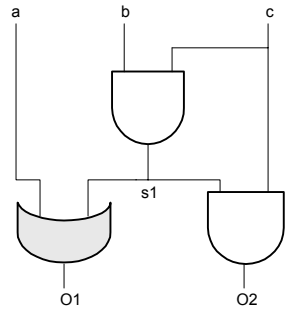


Fig. 1.4: Ejemplo de circuito no árbol

La señal interna s1 es entrada en más de una puerta del circuito, por tanto la puerta de la que es salida será raíz de un árbol. Las puertas que generan O1 y O2 también serán nodos raíz, ya que sus salidas son señales *OUT port*.

Una vez que tenemos el circuito dividido en un conjunto de árboles cuyos nodos raíz están identificados, aplicaremos el algoritmo de árbol sobre cada árbol. Para hacer esto, vamos tomando consecutivamente los nodos raíz del circuito hasta haber aplicado el algoritmo en todos los árboles.

1.4 ALGORITMO DE ÁRBOL.

La forma más sencilla de entender el funcionamiento del algoritmo es ver su aplicación sobre un ejemplo. Vamos a considerar un circuito básico.

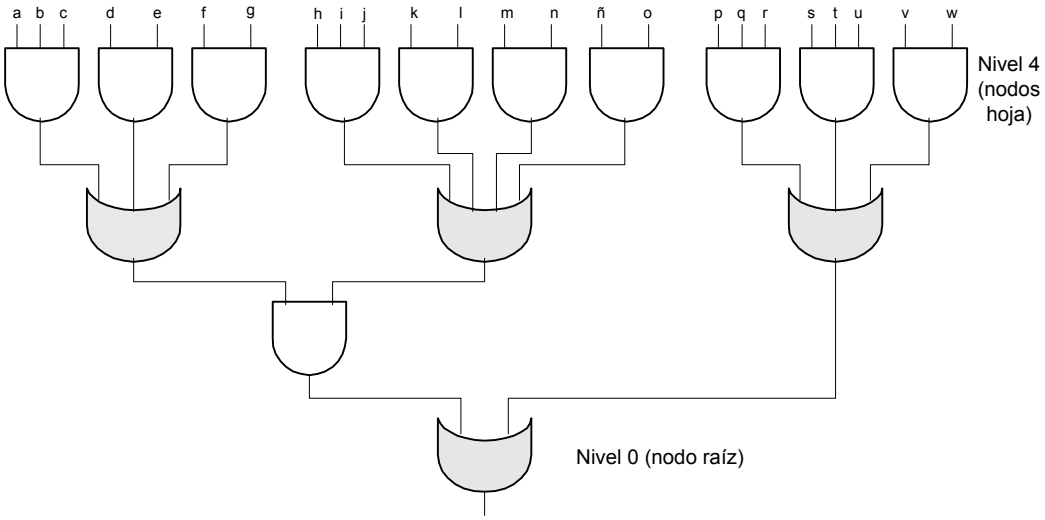


Fig. 1.5: Árbol de 4 niveles

En este ejemplo tenemos un circuito que constituye un árbol, es decir, no contiene bucles. Un árbol se distingue porque la salida de cada puerta entra en una y sólo una puerta del nivel inferior, y así hasta llegar a la raíz. La raíz es única y es la salida global del circuito.

Como veremos más adelante la filosofía del algoritmo es la de reducir circuitos complejos, con bucles, a conjuntos de árboles. La resolución de un circuito árbol mediante el algoritmo, como problema aislado, se realiza de una forma mecánica y puede demostrarse que se llega a una solución óptima para cada árbol.

Diremos que el árbol se divide en niveles. El nivel 0 será el de la puerta raíz, y el nivel máximo el de las puertas hoja o de *fan-in* del circuito. El circuito del ejemplo tiene entonces 4 niveles, contados desde el 0 hasta el nivel 3, donde el nivel 0 es una puerta OR de 2 entradas y el nivel 3 está constituido por un total de 10 puertas.

En un árbol de N niveles el algoritmo comienza a aplicarse tomando cada una de las puertas del nivel N-2 con todas sus puertas entrantes. En el ejemplo tenemos N = 4 y tomaremos los grupos de puertas recuadrados por la línea discontinua en la siguiente figura.

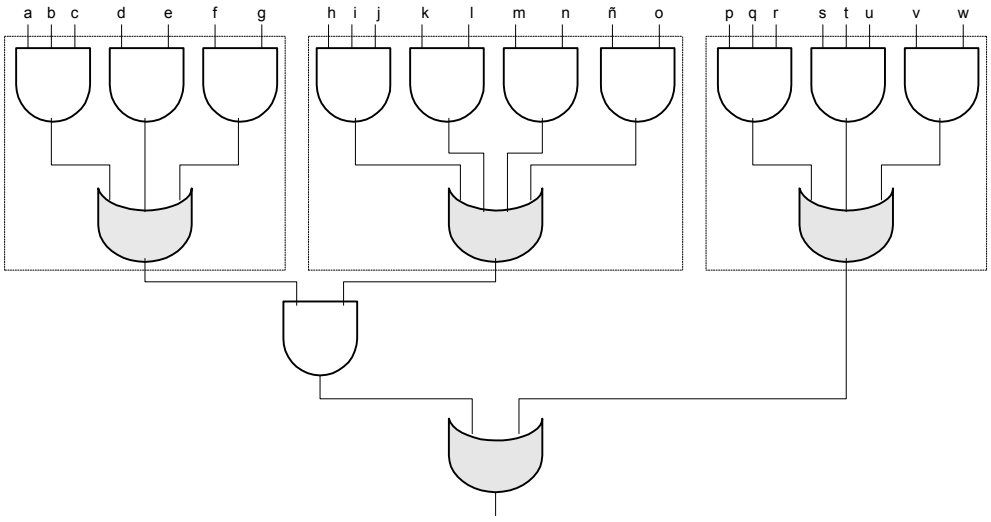


Fig. 1.6: Grupos para el inicio del algoritmo

Empezando de izquierda a derecha, nos quedamos con el primer grupo:

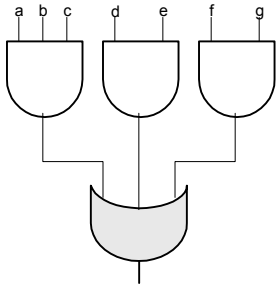


Fig. 1.7: Primer grupo

En este caso las puertas de entrada están ordenadas de izquierda a derecha y de mayor a menor por número de señales de entrada: 3, 2 y 2. De no ser así, la ordenación debe ser el primer paso.

A continuación se trata de ir tomando tantas puertas como sea posible, de izquierda a derecha, sin superar K, el número máximo de entradas de una LUT. En este caso supondremos K = 5, con lo cual podemos tomar las 2 primeras ANDs, siempre por la izquierda. Las agrupamos formando la primera LUT del circuito de la siguiente forma:

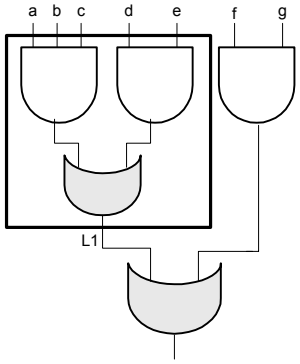


Fig. 1.8: Empaquetado de dos puertas en una 5-LUT

Esta primera LUT que llamaremos L1 queda ya así constituida definitivamente. Sin embargo, con la AND restante de entradas f y g y la puerta OR raíz del subconjunto formaremos una LUT que será no definitiva por el momento, ya que en el siguiente paso puede aparecer la posibilidad de aprovecharla mejor introduciendo más puertas en ella.

Obsérvese que el algoritmo, en su evolución, precisa de la creación de puertas adicionales, como la OR de salida de L1. Es inmediato ver que la lógica que el circuito implementa no cambia:

$$abc + de + fg = (abc + de) + fg$$

Veamos cómo queda el circuito tras finalizar el primer paso del algoritmo en todas las ramas del árbol:

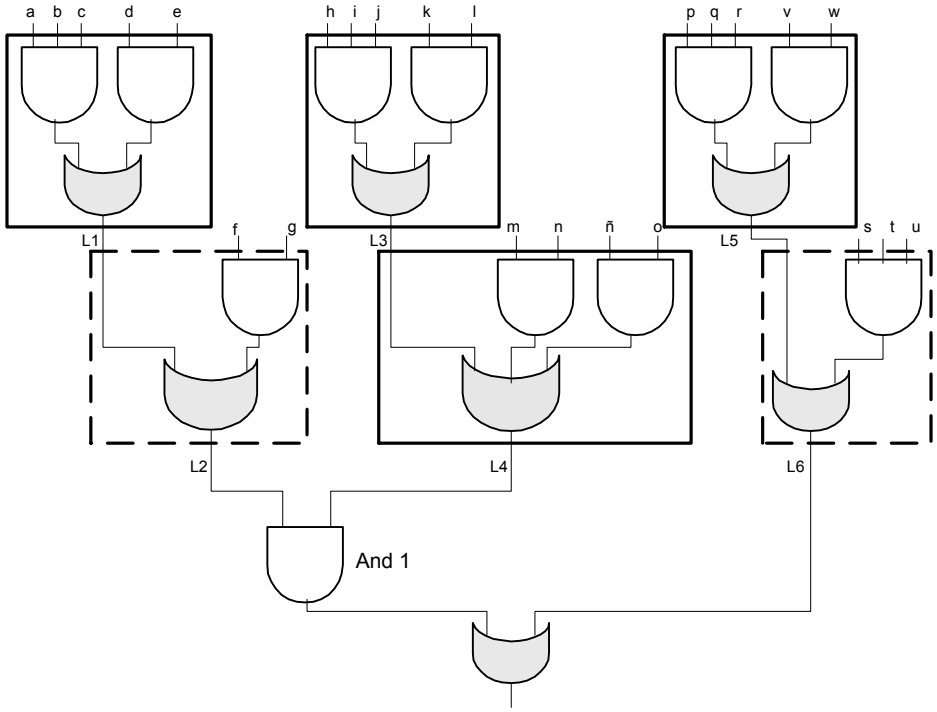


Fig. 1.9: Descomposición multinivel

Las LUTs L2 y L6 no están cerradas definitivamente ya que aún no llegan a tener 5 entradas. Bajamos un nivel en el árbol y nos centramos en la And 1. Se aprovecha que la LUT L2 está aún abierta para incluir a la And 1 en su interior pasando L2 de tener tres a cuatro entradas, por lo que aún así seguirá estando abierta.

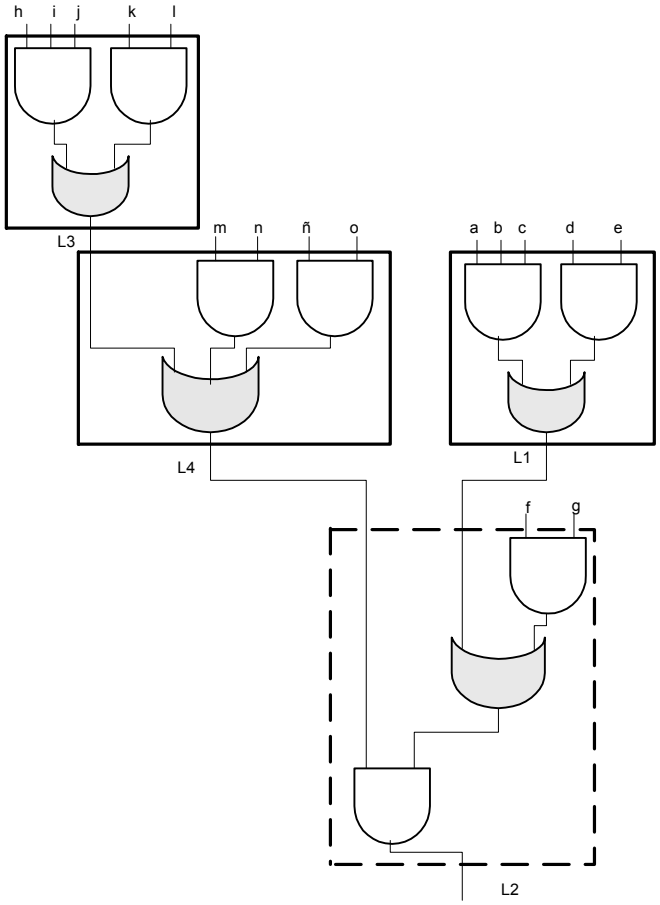


Fig. 1.10: Reordenación y descomposición multinivel

Bajamos al último nivel, donde se encuentra la puerta OR que es el nodo raíz del árbol. Sus dos entradas provienen de L2 y L6, que tienen el mismo número de entradas, cuatro cada una. Por tanto se mantienen en ese orden y la OR raíz se añadirá a L6.

El mapeo final del circuito es el siguiente:

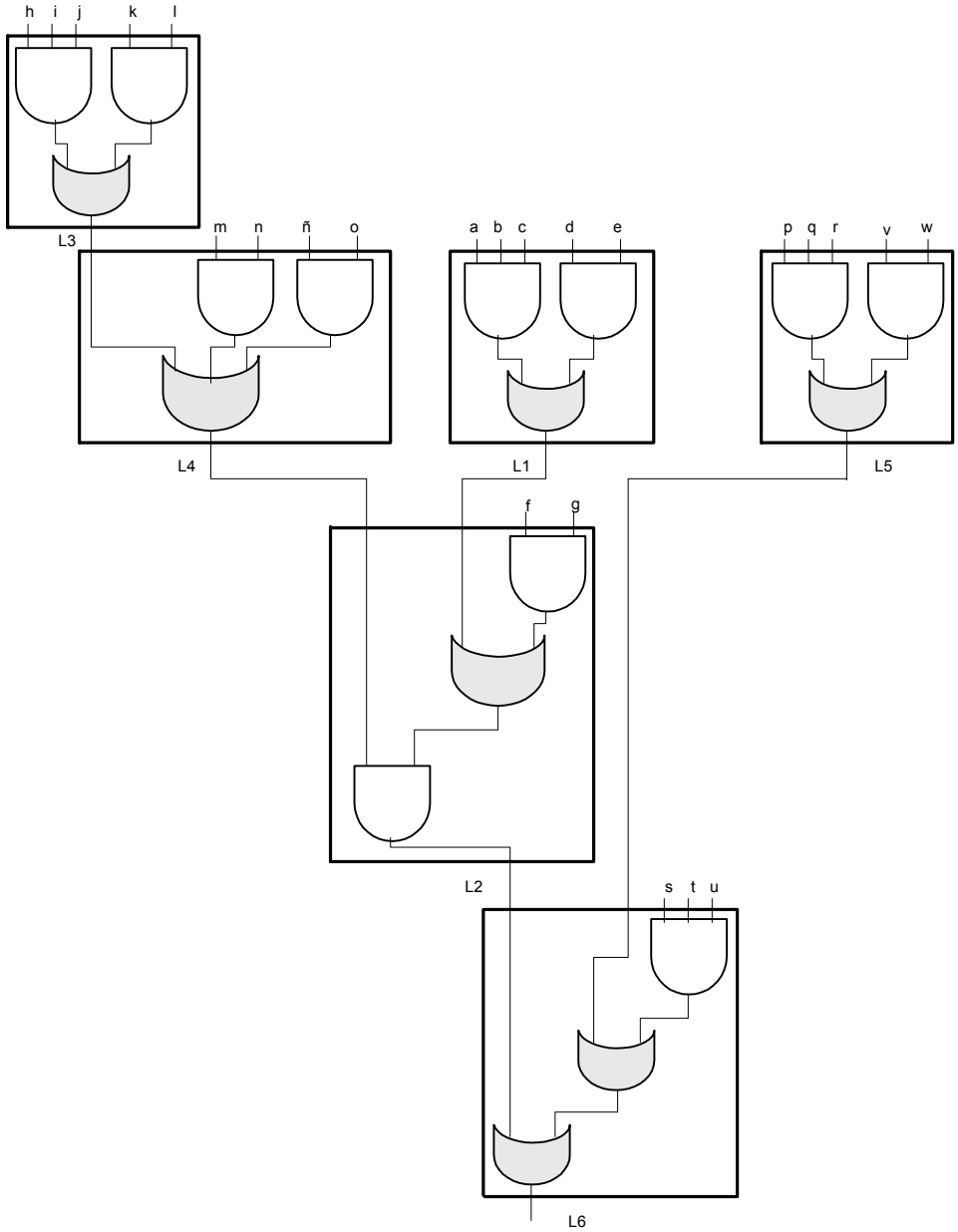


Fig. 1.11: Resultado final del algoritmo

Se ha conseguido mapear el circuito en un total de 6 LUTs, cinco de las cuales tienen el *fan-in* completo, es decir, 5 entradas, y una tiene 4 entradas.

Como ya se ha dicho, la solución a la que hemos llegado mediante la aplicación del algoritmo es óptima por tratarse de un árbol. Aplicando el algoritmo a todos los árboles de un circuito obtendremos la solución óptima para cada árbol, sin embargo no se asegura así que por ello la solución global del mapeo del circuito sea óptima.

Tras la aplicación del algoritmo de árbol a todos los árboles del circuito deberemos seguir optimizando el aprovechamiento de las LUTs mediante mecanismos como la reconvergencia y la replicación. Esto nos llevará a una cuantiosa reducción del número total de LUTs necesarias para el mapeo del circuito, pero después de todo no podremos asegurar que ese número sea el de la solución óptima.

Naturalmente, se espera que dicho número de LUTs sea lo menor posible y que el algoritmo completo obtenga resultados competentes frente a otros en un banco de pruebas con circuitos *benchmarks*.

1.5 OPTIMIZACIÓN POR RECONVERGENCIA.

Se le llama reconvergencia o *fan-out* reconvergente a la aparición de una misma señal de entrada en más de un punto diferente de un circuito. Cuando esto ocurre, que suele ser lo más habitual, es posible aprovecharlo para reducir el número de LUTs necesarias para el mapeo una vez que ya se ha aplicado el algoritmo de árbol.

El procedimiento consiste en buscar las señales repetidas e intentar fusionar aquellas LUTs en las que entran, si es posible hacerlo de forma que en la LUT resultante no se sobrepase el número máximo de entradas K.

Supongamos que tras aplicar el algoritmo de árbol a u circuito obtenemos el resultado de la figura.

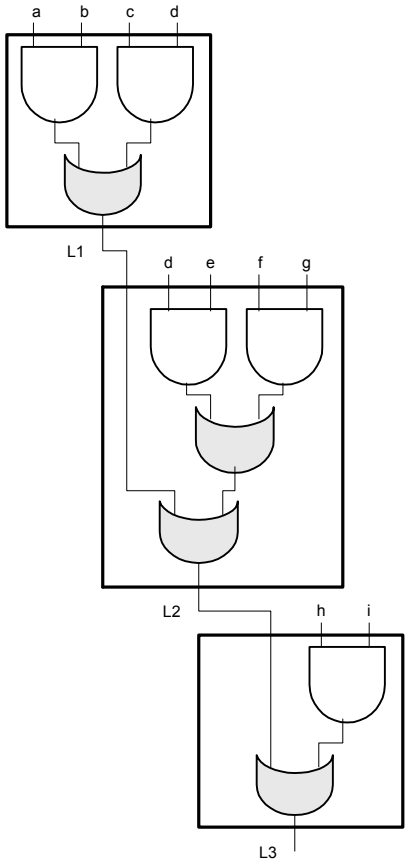


Fig. 1.12: Circuito ejemplo tras aplicar el algoritmo de árbol

Examinando las LUTs una por una, el programa detectará que la señal *d* aparece en dos de ellas, en L1 y L2. Aprovechando esta circunstancia, se realiza la fusión de ambas LUTs para obtener una sola LUT con cinco entradas, lo cual es aceptable para K=5.

Así se reduce en una unidad el número de LUTs necesarias para el mapeo del circuito, que resulta ser el de la figura siguiente.

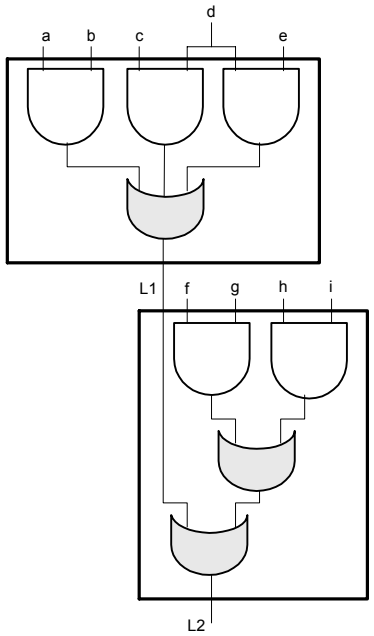


Fig. 1.13: Tras aplicar optimización por reconvergencia

1.6 OPTIMIZACIÓN POR REPLICACIÓN.

La replicación de lógica en nodos de *fan-out* puede reducir el número de LUTs necesarias para implementar un circuito.

En el punto 1.3 se dijo que si una señal interna S es entrada en más de una puerta del circuito, la puerta de la cual S es salida será raíz de un árbol. Esto significa que tras aplicar el algoritmo de árbol, S será la salida de una LUT y a su vez será entrada en más de una LUT.

A veces es posible replicar la lógica necesaria para obtener S introduciendo dicha lógica en cada LUT de *fan-out* donde entre S, ahorrándose así el uso de una LUT.

Sea por ejemplo el circuito de la figura siguiente, mapeado en tres LUTs tras aplicar el algoritmo de árbol.

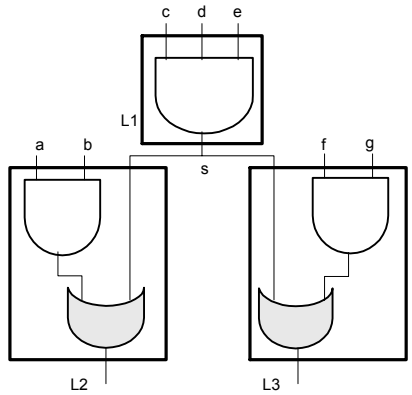


Fig. 1.14: Circuito en 3 LUTs con puerta replicable

Replicando en L2 y L3 la puerta AND que contiene la LUT L1y produce la señal S, nos ahorramos el uso de una LUT, como se ve en la figura siguiente.

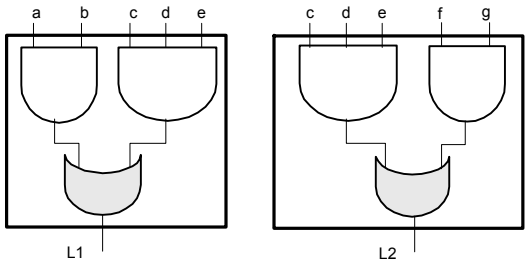


Fig. 1.15: Optimización por replicación: 2 LUTs

2. SEUDOCÓDIGO

En esta sección vamos a exponer la base teórica en la que se fundamentan los algoritmos que hemos visto en los ejemplos del punto anterior mediante seudocódigo y diagramas de flujo.

Así mismo veremos la estructuración del resto de procesos previos a la aplicación del algoritmo, como son la lectura de ficheros de entrada y por último la generación del fichero de salida con formato JLIF.

2.1 FUNCIÓN PRINCIPAL: LECTURA DE LOS FICHEROS DE ENTRADA

En la función principal (*main*) del programa se abren para su lectura el fichero VHDL de entrada y el fichero de librería. Los objetivos de esta función son:

- Identificar cada uno de los elementos del circuito (puertas y FFs), reservar memoria y almacenarlo en RAM mediante una estructura de datos.
- Crear una lista enlazada con todos los elementos del circuito para hacer posible el acceso a los mismos (cada uno representado por una estructura de datos).
- Identificar las relaciones de entrada/salida ente los elementos creando árboles de estructuras.
- Recorrer todos los árboles en que se descompone el circuito y aplicar a cada uno los algoritmos de mapeo y optimización para obtener árboles de LUTs.
- Escribir el fichero JLIF de salida leyendo los árboles de LUTs resultantes.

Para realizar cada una de estas tareas, la función principal llama a una serie de funciones, muchas de las cuales son recursivas ya que están pensadas para trabajar con árboles de estructuras de datos.

La lectura de los ficheros de entrada, reserva de memoria y construcción del circuito en RAM son tareas rutinarias más pesadas que complejas en el desarrollo, y no considero necesario entrar más en detalle.

Las funciones más complejas y que revisten mayor interés son las que aplican los algoritmos de mapeo y optimización a cada nodo del circuito. Veremos a continuación seudocódigo para estas funciones.

2.2 SEUDOCÓDIGO PARA EL ALGORITMO DE ÁRBOL

Una vez que tenemos el circuito separado en una serie de árboles de puertas lógicas o nodos, se irá recorriendo cada nodo de cada árbol para aplicarle el algoritmo de mapeo de árbol.

El siguiente seudocódigo se aplica a cada nodo junto con sus LUTs de Fan-in, es decir, aquellas cuya salida sea entrada del mismo.

```
MapeoDeArbol (nodo, LUTsDeFan-in)
{
        ordenar LUTsDeFan-in de mayor a menor k y de izquierda a derecha
        N = Nº de LUTsDeFan-in
        para i=1 hasta N                        /* Descomposición de dos niveles */
        {
                para j=1 hasta i-1
                {
                        si número de entradas (LUT_i + LUT_i)  <= K
                                Empaquetar (LUT_i ,LUT_j)
                }
```

```
}

ordenar LUTsEmpaquetadas de mayor a menor k y de izda. A dcha.
N = Nº de LUTsEmpaquetadas
para i=1 hasta N                        /* Descomposición multinivel */
{
        para j=i+1 hasta N
        {
                si número de entradas LUT_j + 1 <= K
                        Reubicar (LUT_i, LUT_j)
                reordenar LUTsEmpaquetadas
                salir del bucle j
        }
}
```

Fig. 2.1: Seudocódigo para el algoritmo de árbol

La aplicación de este algoritmo debe empezar por los nodos superiores de cada árbol. No debe aplicarse a un nodo de un nivel N hasta que no se haya aplicado a todos los nodos de nivel mayor que N. El último nodo será siempre el nodo raíz del árbol, al que consideramos de nivel 0.

A las puertas lógicas de mayor nivel de un árbol se les llama *puertas hoja*. Cuando comienza la aplicación del algoritmo en cada árbol creamos una LUT para cada puerta hoja que la contiene a ella únicamente. Entonces comenzamos a aplicar el algoritmo visto en el seudocódigo anterior por los nodos donde entren las salidas de las puertas hojas, de izquierda a derecha en toda la parte de mayor nivel del árbol. El proceso se va repitiendo siempre bajando un nivel cada vez hasta llegar al nodo raíz. Entonces podemos dar por finalizado el algoritmo de árbol.

En el listado de la figura 2.1 podemos ver que el algoritmo tiene dos fases: descomposición de dos niveles y descomposición multinivel.

Dentro de la descomposición de dos niveles tenemos la función Empaquetar(LUT_i ,LUT_j). Esta función toma dos LUTs y las funde en una sola uniendo sus salidas en una puerta de dos entradas, la cual será la salida de la nueva LUT empaquetada. Es lo que vimos en el ejemplo de las figuras 1.7 y 1.8 con las dos primeras puertas.

En la descomposición multinivel destaca la función Reubicar(LUT_i ,LUT_j). Es lo que vimos en la figura 1.9 donde la LUT L1 pasa a ser entrada en la L2, y ocurre lo mismo con L3 en L4 y L5 en L6. Reubicando todas las LUTs de entrada a un nodo, lo que antes eran dos niveles ahora pasan a ser tantos como LUTs hubiera en el nivel superior.

2.3 SEUDOCÓDIGO PARA OPTIMIZACIÓN: RECONVERGENCIA Y REPLICACIÓN

En este proyecto, la explotación de la reconvergencia para reducir la ocupación de área se realizó de forma conjunta con la aplicación del algoritmo de árbol. Durante la descomposición de dos niveles, se consideran como una sola las señales de entrada repetidas a la hora de empaquetar dos LUTs.

Esto requiere simplemente la modificación de la función de empaquetamiento de LUTs que vimos en el apartado anterior, y no precisa de ningún otro algoritmo posterior al de árbol.

La optimización por replicación de lógica sí es posterior a la aplicación del algoritmo de árbol a todo el circuito. El algoritmo para replicación se basa en el seudocódigo de la figura 2.2:

```
N = Nº de LUTs del circuito que son raíz de un árbol (LUT-Raíz1 ... LUT-RaízN)
para i=1 hasta N
{
        M = Nº de LUTs donde entra la salida de LUT-Raízi (LUT-Hoja1 ... LUT-HojaM)
        para j=1 hasta M
        {
                Si Nº de entradas de LUT-Raízi + Nº de entradas LUT-Hojaj > K
                        Marcar LUT-Raízi como No-Replicable
        }
}
```

Fig. 2.2: Algoritmo de replicación

Al aplicar esto a todas las LUTs raíz de los árboles del circuito ya sabremos cuáles de ellas son no replicables y cuáles lo son y en cuántas LUTs hoja cada una. Entonces replicamos empezando por aquellas que requieran replicación en menos LUTs hoja, ya que así es menor la probabilidad de que estropeen otras posibilidades de replicación.

Nótese que tras cada replicación hay que volver a aplicar el algoritmo de la figura 2.2, ya que la topología cambia.

3. DESARROLLO DEL CÓDIGO FUENTE

En esta sección veremos en detalle los aspectos técnicos relativos al desarrollo del programa de mapeo. Es en este punto donde comienza realmente el desarrollo material del proyecto en cuanto a que damos por suficiente la investigación y planificación realizada y pasamos a la escritura del código.

Empezaremos viendo en los siguientes apartados el lenguaje y entorno de programación elegidos para el desarrollo, para continuar con las estructuras y tipos de datos principales. Por último veremos la estructura general del programa y destacaremos las funciones más relevantes.

3.1 LENGUAJE DE PROGRAMACIÓN Y ENTORNO DE DESARROLLO

El desarrollo de un algoritmo complejo como es el caso del algoritmo de mapeo que nos ocupa presenta una elevada dificultad en la programación, principalmente por el tamaño, la complejidad y porque la variedad de circuitos que se pueden presentar como entrada es muy amplia.

La forma adecuada de abordar este tipo de problemas es utilizando estructuras de datos y topologías de conexión en listas enlazadas y árboles. El lenguaje más apropiado en este caso es el lenguaje C, el cual brinda un gran control en la definición de estructuras de datos y sobre las referencias entre las mismas mediante punteros. Además permite la creación de las estructuras necesarias mediante reserva dinámica de memoria, y como es sabido es un lenguaje muy eficiente en cuanto a uso de memoria y tiempo de proceso.

En cuanto al entorno de programación he elegido Borland C++ 5.0, un entorno ampliamente contrastado, muy completo y fiable, que además ofrece una extensa librería de ayuda y ejemplos que puede ser de enorme utilidad en cualquier circunstancia.

Este entorno permite, entre otras posibilidades, crear aplicaciones Windows visuales en C++, que se ejecutan en una ventana y crear aplicaciones ejecutables (.exe). En el caso que nos ocupa usaremos el tipo EasyWin para crear un ejecutable, sin necesidad de crear un proyecto de tipo Windows, con ventanas, ya que este programa está orientado finalmente a ser incluido dentro del conjunto de herramientas CAD del proyecto FIPSOC.

3.2 ESTRUCTURAS DE DATOS

Se han utilizado únicamente dos estructuras de datos para todo el programa: una destinada a almacenar los datos de cada puerta lógica del circuito y otra para cada biestable o Flip-Flop.

En la figura 3.1 se muestra el código asociado a la definición de la estructura que alberga las puertas lógicas. Los comentarios que aparecen explican por encima el significado de los campos que la componen.

```
struct PUERTA {
        int espr;                   //1 si es el primer elto. de la lista enlazada de puertas
        char dev[LNG];              //nombre del dispositivo en el fichero de librería
        char entr[5][LNG];          //nombres de las señales de entrada
        int inlev[5];               //será 1 si la entrada i tiene un nivel superior (i=0 ... 4)
        struct PUERTA *sig;         //puntero al siguiente elto. en la lista enlazada
        char nombre[LNG];           // nombre de la señal de salida de esta puerta
        int neg;                    //1 si la salida es negada
        int esroot;                 //1 si esta puerta es el nodo raíz de un árbol
```

```
    int nrep;                   //nº de veces que se puede replicar

    int tipo;                   // 0->OR,1->AND, 2->fan-in port ,4->XOR
    int nent;                   // nº de entradas
    int nfan;                   //nº de entradas de fanin
    int nlut;                   //nº de LUT al que pertenece (inicialmte 0)

    int lut;                    //0->nada,1->provisional,2->definitiva,3->a extinguir
    int tot;                    //nº total de entradas ocupadas en fanin
    unsigned long func;         //función de salida COMB_TILE (5-LUT)
    unsigned int func2;         //funcion de salida 4-LUT
    struct PUERTA *fan[5];      // array de 5 punteros a FANINs de entrada
    struct PUERTA *ent[5];      // array de 5 punteros a PUERTAs de entrada
    int negfan[5];              //1 si la entrada i es negada (i=0 ... 4)
    int negent[5];
    };
```

Fig. 3.1: Definición de la estructura que alberga una puerta lógica

Por otra parte, para almacenar los biestables tenemos la estructura de la figura 3.2. Esta estructura es mucho más reducida, puesto que este programa no realiza ningún tipo de procesamiento con los FFs, simplemente se requiere almacenarlos para enumerarlos en el fichero JLIF de salida con su función asociada correspondiente.

```
struct FF {
    char salida[9][LNG];        //señales de salida
    struct FF *sigff;           //puntero al siguiente FF en la lista enlazada
    } ;
```

Fig. 3.2: Definición de la estructura para los FFs

A continuación veremos cómo se instancian estas estructuras en la interfaz de entrada del programa, y cómo se inicializan algunos de sus campos. Con ello explicaré la forma en la que el circuito que describe el fichero VHDL de entrada es leído y se crea una imagen del mismo en memoria.

3.2.1 Interfaz de entrada

La interfaz de entrada del programa se encarga de leer el fichero VHDL de entrada, y reservando memoria dinámicamente crea una instancia de esta estructura por cada puerta lógica que aparece.

Si examinamos el fichero de librería "fipsoc.gl", que aparece en el anexo 1, vemos que algunos elementos se componen de más de una puerta lógica, como es el caso por ejemplo de *aoi21* o *mux21*. Cuando aparecen, este tipo de elementos se dividen en puertas lógicas AND, OR y XOR, negadas o no, según indique la función en el fichero de librería. Entonces se crea una estructura para cada puerta componente y se inicializan los campos correspondientes para almacenar la información de las conexiones entre ellas.

Las puertas se van instanciando según se leen del fichero VHDL, y mediante el puntero

```
struct PUERTA *sig;
```

se almacenan todas las puertas lógicas del circuito en una lista enlazada simple, donde cada estructura de puerta almacena el número de entradas que tiene y los nombres de todas ellas más el de la señal de salida y si están negadas o no, etc..

Esta información es suficiente para que a continuación una función recorra esta lista de puertas y mediante el array de punteros

```
struct PUERTA *ent[5];
```

establezca las conexiones entre las puertas formando árboles de estructuras puerta.

Tras esto ya tendremos en memoria todo el circuito lógico y su estructura, listo para la aplicación del algoritmo.

3.3 DEFINICIONES DE CONSTANTES Y VARIABLES GLOBALES

El programa utiliza una serie de definiciones de constantes, que en lenguaje C se enuncian con la palabra clave "#define". He usado estas definiciones para dimensionar las capacidades del programa, indicando por ejemplo el máximo nº de inversores que habrá en el circuito VHDL de entrada, el máximo nº de señales de salida globales o la máxima longitud de los nombres de las señales en número de caracteres.

Los valores de estas constantes se han tomado de forma que sean de sobra suficientemente grandes para el mayor posible circuito que se pueda mapear en una FPGA del proyecto FIPSOC. En realidad el programa admitiría circuitos de mucho más de 10.000 puertas sin ningún problema, muy por encima del tamaño de los circuitos *benchmarks*.

En cuanto a variables globales, atendiendo a los principios de la programación estructurada, he utilizado las indispensables. Como veremos más adelante, el programa se compone de un gran número de funciones recursivas. Las variables globales son necesarias sobre todo para almacenar contadores generales. Destacan las siguientes:

```
int nluts;          //contador del total de LUTs necesarias para el circuito
int ntent,ntsal;    //contador del número total de entradas y salidas globales
int ninv;           //contador del número de inversores en el circuito
```

Es una buena práctica el uso del mínimo posible de variables globales. No obstante a veces es conveniente valorar su uso para evitar que aumente excesivamente el número de parámetros que se pasa a las funciones.

Demasiados parámetros pueden ser problemáticos en cuanto a rendimiento y en cuanto a la claridad del código, sobre todo cuando muchas de las funciones son recursivas y se complica la llamada a las mismas.

3.4 LA FUNCIÓN PRINCIPAL "main()"

En un programa escrito en lenguaje C siempre hay una función principal llamada función *main()*. Esta es la primera función que se ejecuta al arrancar el programa, y es desde donde se va llamando al resto de procedimientos. Una función *main* no debe ser excesivamente larga, según recomiendan los principios de la programación estructurada. El programa debe estar correctamente modularizado en funciones que se encarguen del proceso de datos y que se vayan llamando desde la función principal.

La función *main* de este programa de mapeo realiza una serie de acciones y llamadas a procedimientos:

- Declaración de variables locales, entre las que destacan dos arrays de cadenas de caracteres destinados a almacenar los nombres de las señales de entrada y de salida del circuito.
- Reserva dinámica de memoria para dichos arrays.
- Apertura del fichero de librería "fipsoc.gl" y del fichero de entrada cuyo nombre ha de ser introducido por el usuario, "*.vhd".
- Creación del fichero de salida, cuyo nombre también deberá ser introducido por el usuario, "*.jli".
- Llamada sucesiva a las funciones con diferentes cometidos:

- Lectura del fichero VHDL de entrada y almacenaje del circuito en memoria en forma de lista de árboles de puertas lógicas.
- Aplicación del *algoritmo de árbol* a todos los árboles del circuito.
- Optimización por *replicación*.
- Escritura del fichero de salida en formato JLIF.

- Se muestran mensajes al usuario informando de la finalización del procesado y se cierran los ficheros.

Las funciones mencionadas a las que se llama desde esta función principal son funciones recursivas, pensadas para trabajar sobre árboles de estructuras. En el punto siguiente veremos más afondo su funcionamiento y el paso de parámetros que se realiza en la llamada.

3.5 FUNCIONES RECURSIVAS

El algoritmo de mapeo en este programa está desarrollado fundamentalmente mediante funciones recursivas aplicadas sobre árboles de estructuras.

Estas funciones están diseñadas para aceptar como parámetro un puntero a una estructura de datos, la cual contiene una matriz de punteros a estructuras similares, de forma que es posible recorrer todo el árbol partiendo del nodo raíz.

Pero en este programa no se trabaja con un solo árbol, sino con un conjunto o *bosque* de árboles, los cuales se relacionan entre sí mediante los respectivos nodos raíz, ya que forman una lista enlazada. Cada nodo raíz contiene un puntero a otro nodo análogo, exceptuando al último de la lista, que tendrá este campo vacío mientras no se añadan más árboles.

En la figura 3.3 encontramos un esquema que representa la organización de los nodos o estructuras de datos en las que se basa este programa.

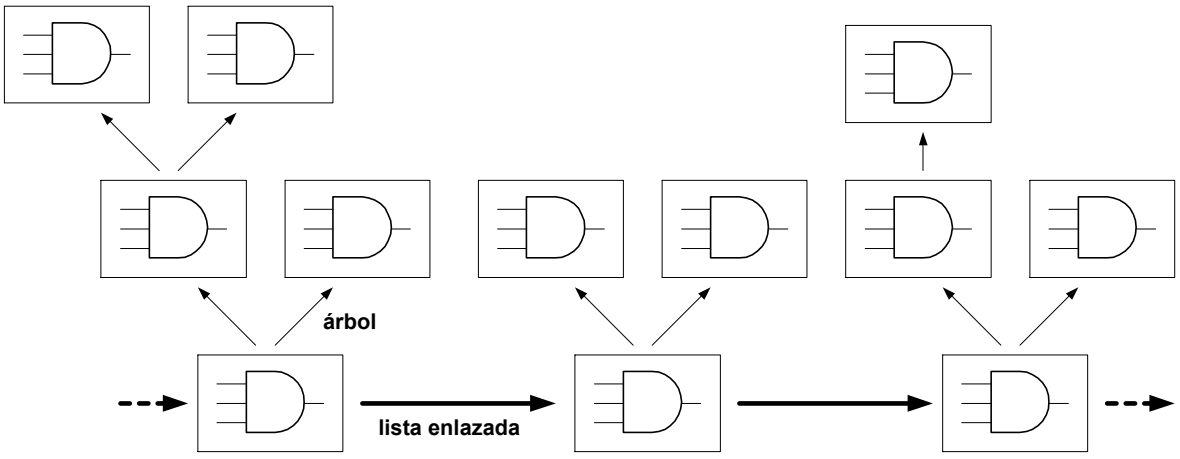


Fig. 3.3: Lista enlazada de nodos raíz de árboles

En este gráfico cada caja representa una estructura en la que se almacenan los datos de una puerta lógica, y las flechas son punteros que relacionan las estructuras entre si.

3.5.1 Algoritmo de árbol

El algoritmo de mapeo está desarrollado mediante dos funciones: una que va recorriendo la lista enlazada de nodos raíz, llamando según sea necesario a una segunda función denominada "fanin", que será recursiva, y pasándole como parámetro la estructura correspondiente a un nodo raíz de un árbol. Esta función recursiva es la que realiza el proceso del algoritmo de árbol sobre el árbol en cuestión, según el seudocódigo que ya se vio en la figura 2.1.

A continuación veremos el código correspondiente a esta función acompañado de comentarios aclaratorios sobre su funcionamiento. Esta es la función clave del programa, aunque en ella aparecen llamadas a funciones menores, y lleva a cabo el mapeo tecnológico de un árbol.

```
void fanin(struct PUERTA* act)          //recibe como parámetro una estructura PUERTA
{
        /*DESCOMPOSICIÓN DE DOS NIVELES*/

struct PUERTA* nuevo;
struct PUERTA* grande;                  //puntero al la LUT de entrada + grande (con + entradas)
int n,mx,k,fin,nsel,sum,grneg,rp;
int eleg[5]={0,0,0,0,0};
int suma[5]={0,0,0,0,0};
char lista[LUTK][LNG];
char strg[LNG];

for (n=0;n<act->nent;n++) {             //recorre las entradas de esta puerta
        if (act->ent[n]->lut==0 && (act->ent[n]->tipo!=CABLE))
                fanin(act->ent[n]);     //llamada recursiva pasando la entrada n como parámetro
        if (act->ent[n]->lut==3) act->ent[n]=act->ent[n]->fan[act->ent[n]->nfan];
act->lut=1;
act->nfan=act->nent;
k=0;
for (mx=klut;mx>0;mx--)                 //si todas las entradas son fanin
        for (n=0;n<act->nent;n++)       //recorre las entradas
                if (act->ent[n]->tot==mx) {
                        act->fan[k]=act->ent[n];    //las ordena de > a < nº total de entradas
                        act->negfan[k]=act->negent[n];
                        k++; }
k=0;
fin=0;
while (!fin)
        {
        sum=0;
        nsel=0;
        for (n=k;n<act->nfan;n++)
                {
                rp=0;
                if (eleg[n]==0) rp=pack(sum,lista,act->fan[n]); //llama a la función empaquetar
                if (rp)
                        {
                        sum=rp;
                        eleg[n]=1;
```

```
                        suma[nsel]=n;
                        nsel++;          //incrementa nº de entradas fanins seleccionadas
                        }
                } //fin for
        if (nsel>1)
                {
                nuevo=(struct PUERTA*)malloc(sizeof(struct PUERTA));
                if (nuevo == NULL) error(5);
                nuevo->tipo=act->tipo;              //cuando empaquetamos puertas se
                nuevo->nent=nsel;                        //crea una nueva
                nuevo->nfan=nsel;
                nuevo->lut=1;
                nuevo->tot=sum;
    strcpy(nuevo->nombre,"s");
                strcat(nuevo->nombre,itoa(sens,strg,10));   //es necesario crear un nombre
                strcat(nuevo->nombre,"_");                  //de señal no repetido
                strcat(nuevo->nombre,act->nombre);
                sens++;
                for (n=0;n<nsel;n++) {
                        nuevo->fan[n]=act->fan[suma[n]];
                        nuevo->negfan[n]=act->negfan[suma[n]]; }
                act->fan[k]=nuevo;
                act->negfan[k]=0;
                k++;
                }
        else if(nsel==1) {
                act->fan[k]=act->fan[suma[0]];
                act->negfan[k]=act->negfan[suma[0]];
                k++; }
        else if (nsel==0) fin=1;      //si no coges ninguna es porque no quedan
        }   //fin del while(!fin)
act->nfan=k;                    //k packs ubicados en act->fan[0] hasta act->fan[k-1] ordenados
for (k=0;k<act->nfan-1;k++)     //ordenamos por tot antes del decomp
        {
        mx=act->fan[k]->tot;
        for (n=k+1;n<act->nfan;n++)
                {
                if (act->fan[n]->tot>mx)
                        {
                        mx=act->fan[n]->tot;
```

```
                        rp=n;
                        }
                }
        if (mx>act->fan[k]->tot)
                {
                grande=act->fan[rp];
                act->fan[rp]=act->fan[k];
                act->fan[k]=grande;
                }
        }

        /*DESCOMPOSICIÓN MULTINIVEL*/

while (act->nfan>1)
        {
        grande=act->fan[0];     //coge el + grande
        grneg=act->negfan[0];
        grande->lut=2;          //será lut definitiva
        nluts++;
        grande->nlut=nluts;
        act->nfan--;
        for (n=0;n<act->nfan;n++) {
        act->fan[n]=act->fan[n+1];      //todos corren a izda.
        act->negfan[n]=act->negfan[n+1]; }
        nsel=0;
        while (act->fan[nsel]->tot==klut && nsel<act->nfan) nsel++;     //toma el 2º mayor
        if (nsel==act->nfan)    //pero si no lo hay
        {
                nuevo=(struct PUERTA*)malloc(sizeof(struct PUERTA)); //creamos uno vacío
                if (nuevo == NULL) error(5);
                nuevo->tot=1;
                nuevo->nfan=1;
                nuevo->tipo=act->tipo;
                nuevo->lut=1;
                act->fan[nsel]=nuevo;
                nuevo->fan[0]=grande;
                nuevo->negfan[0]=grneg;
                act->nfan++;
    strcpy(nuevo->nombre,"s");
                strcat(nuevo->nombre,itoa(sens,strg,10));
```

```
                strcat(nuevo->nombre,"_");
                strcat(nuevo->nombre,act->nombre);
                sens++;
                }
        else {    //si lo hay
        nuevo=(struct PUERTA*)malloc(sizeof(struct PUERTA));
        if (nuevo == NULL) error(5);
        nuevo->nfan=2;
        nuevo->tot=1+act->fan[nsel]->tot;
        nuevo->tipo=act->tipo;
        nuevo->lut=1;
        nuevo->fan[0]=grande;
        nuevo->negfan[0]=grneg;
        nuevo->fan[1]=act->fan[nsel];
        nuevo->negfan[1]=act->negfan[nsel];
        act->fan[nsel]=nuevo;
        act->negfan[nsel]=0;
  strcpy(nuevo->nombre,"s");
        strcat(nuevo->nombre,itoa(sens,strg,10));
        strcat(nuevo->nombre,"_");
        strcat(nuevo->nombre,act->nombre);
        sens++;
        }
        } //fin del while (act->nfan>1)
act->lut=3;
act->nfan=nsel;
} //fin de fanin
```

Fig. 3.4: Código para el algoritmo de árbol

3.5.2 Interfaz de salida

Me parece interesante destacar que la interfaz de salida, es decir, el código que se encarga de generar el fichero de salida en formato JLIF está también basado en una función recursiva.

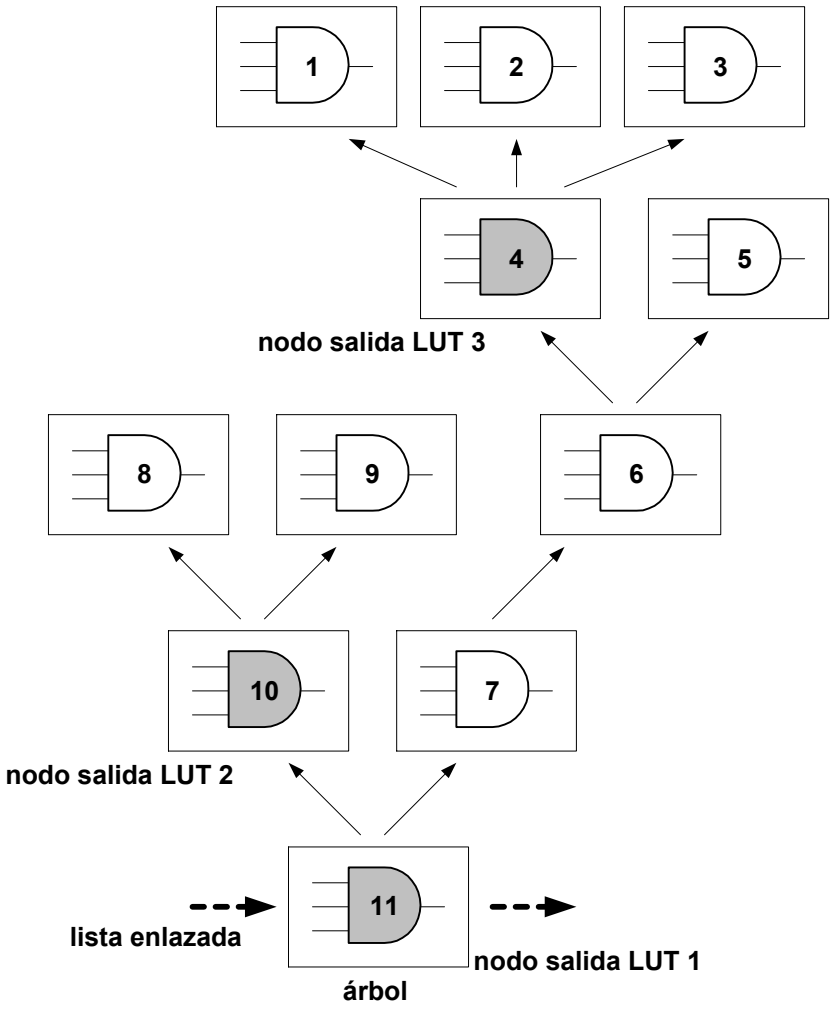
Lo que se ha hecho mediante el algoritmo de árbol y la posterior optimización es manipular, crear y destruir puertas lógicas del circuito, alterando también las conexiones. Pero nótese que no se ha definido una estructura de datos particular que albergue una LUT.

Una LUT se puede ver como un conjunto de puertas lógicas conectadas entre sí con una única salida. De modo que si en un árbol marcamos aquellas puertas que son nodo de salida de una LUT, esto será suficiente para conocer completamente la división del circuito en LUTs.

Esto es lo que se hace en el código mediante el campo *lut* de la estructura de datos *PUERTA*. Como ya se vio en la definición de la estructura de la figura 3.1, este campo indica para cada puerta, según su valor, si es o no el nodo de salida de una LUT, y si lo es indica si es una LUT provisional (no está llena) o definitiva.

Al término del algoritmo todas las LUTs serán definitivas, y para obtener el fichero de salida se usa un mecanismo análogo al visto en el apartado anterior: una función recorre la lista enlazada, llamando a otra función para cada nodo raíz que recorre cada árbol mediante recursividad. Esta segunda función reconoce donde empieza y termina cada LUT y calcula su función de salida.

En la figura 3.5 podemos ver la conformación de las estructuras de datos de un circuito a la hora de obtener el fichero JLIF mediante la interfaz de salida.



Los nodos marcados en color gris son las puertas de salida de las LUTs del circuito. Como se puede comprobar, esta información es suficiente para conocer a que LUT pertenece cada puerta y la función de salida de cada LUT.

En este caso podemos ver que las puertas 1, 2, 3 y 4 constituyen la LUT 3, las puertas 8, 9 y 10 constituyen la LUT 2 y el resto de puertas pertenecen a la LUT 1.

3.6 MEMORIA Y TIEMPO DE EJECUCIÓN

En este apartado vamos a evaluar las necesidades de capacidad de memoria RAM que tiene el programa, y además veremos las consideraciones que se han realizado al elegir la estrategia de programación de tal manera que el tiempo de ejecución sea el menor posible.

3.6.1 Requisitos de memoria

Los requisitos de memoria vienen fijados por el tamaño del circuito VHDL de entrada: se necesitará una estructura PUERTA (ver figura 3.1) por cada puerta lógica del circuito de entrada y una estructura FF (ver figura 3.2) por cada biestable.

Suponiendo que mantenemos la configuración inicial de 50 caracteres de máximo para las cadenas que se manejan en el programa, como por ejemplo los nombres de las señales, el tamaño total que ocupa una estructura PUERTA en el sistema en el que se ha desarrollado el código es de 418 bytes. Este tamaño puede variar en otros sistemas. Para la estructura FF el tamaño es de 454 bytes.

3.6.2 Tiempo de ejecución

El circuito benchmark de mayor tamaño con el que se ha probado el programa tiene un total de 2273 puertas lógicas, lo que supondrá 928 Kb. Teniendo en cuenta que el programa requiere el uso de otras variables y cadenas de caracteres, el espacio requerido total de memoria RAM es del orden de un megabyte (1 Mb).

En cuanto al tiempo de ejecución se probaron dos estrategias diferentes. Uno de los tipos de operaciones que más tiempo de CPU suele consumir es el manejo de cadenas de caracteres (copia, comparación de dos cadenas, etc.). Especialmente en este programa, donde se trata con nombres de señales que a veces pueden ser bastante largos, ya que provienen de la fase anterior al *place & route* (ver introducción, figura I.1). Por ello puse especial atención en estas operaciones a la hora de depurar el programa y mejorar el tiempo de ejecución.

Se elaboraron dos versiones diferentes del programa:

- La primera sencillamente trabaja con cadenas, usando entre otras las funciones *strcmp* para comparaciones y *strcpy* para copias.

- La segunda almacena todas las cadenas de caracteres que aparezcan en una matriz de forma que a cada una le corresponde un número (según el orden). Se espera que el uso de números en lugar de cadenas mejore el tiempo de ejecución, ya que las comparaciones y copias de números son más rápidas.

En la fase de pruebas se demostró que la primera alternativa era mucho mejor que la segunda, debido a que el tiempo que se gana en las copias y comparaciones de cadenas se pierde con creces almacenando estas cadenas en la lista.

El tiempo de ejecución para el mayor fichero VHDL del banco de pruebas, con 2273 puertas lógicas, fue de 7'34 segundos en un PC Pentium III a 500 MHz.

V. FASE DE PRUEBAS

FASE DE PRUEBAS

El programa ha sido ampliamente probado utilizando un banco de circuitos benchmarks conocido, el cual supone un patrón de medida estándar para los algoritmos de technology mapping.

Para tener una referencia de comparación que nos permita valorar los resultados, se han realizado las mismas pruebas con el programa comercial *SIS* de forma simultánea.

En las tablas siguientes se muestran los resultados para K=4. En la primera de ellas los circuitos de entrada son puramente combinacionales, mientras que en la segunda los circuitos contienen diversos tipos de biestables como elementos secuenciales.

Fichero VHD	Nº de puertas	SIS con K=4 Opción LOW	SIS con K=4 Opción HIGH	FIPMAP con K=4
SEQ	2273	1823	1814	1822
ALU4	1289	1091	1087	1081
APEX2	191	122	119	125
5EXP1	71	43	37	43
CLIP	164	109	109	109
MM9B	205	103	102	101

MULT16B	139	85	74	85
S298	1605	770	767	769
TBK	111	73	70	73
MULTIP2	648	204	204	224
TOTAL	6696	4423 (+40)	4383	4432 (+49)

Tabla 1: Resultados con circuitos combinacionales para K=4

Como se puede observar, los resultados en los dos programas son bastante parejos para la opción LOW de SIS (baja optimización), y ligeramente superiores en SIS con la opción HIGH. Aunque la diferencia de número de LUTS se concentra principalmente en el fichero VHDL "multip2", que es donde FIPMAP obtiene los peores resultados.

Aún así, la diferencia es de tan solo 49 LUTs sobre un total de 6696 puertas lógicas, siendo la mayor diferencia de 20 LUTs sobre un circuito de 648 puertas lógicas.

Destacar también que en el circuito "alu4", de 1289 puertas, FIPMAP obtiene un resultado mejor que SIS en 6 puertas, y en "mm9b" es también mejor por 1 puerta.

Pero sobre todo, el punto a favor de FIPMAP es el tiempo de ejecución, que como se vio en la sección anterior es de poco más de 7 segundos para el mayor circuito, mientras que el programa SIS tarda más de 15 minutos en resolver el mapeo con la opción HIGH.

Fichero VHD	Nº de puertas	SIS con K=4 Opción LOW	SIS con K=4 Opción HIGH	FIPMAP con K=4
SPLA	1691	1204	1197	1212
BEECOUNT	28	17	16	17
MISEX3C	275	184	181	186
EX1010	1591	1154	1150	1165
PDC	294	195	195	196
TOTAL	3879	2754 (+15)	2739	2776 (+37)

Tabla 2: Resultados para K=4 con circuitos secuenciales

Como se puede comprobar en la tabla 2, los resultados de las pruebas con circuitos secuenciales son parecidos en cuanto a nº de LUTs a los obtenidos con circuitos puramente combinacionales.

La mayor diferencia es de 15 LUTs en un circuito de 1691 puertas, y en total la diferencia es de 37 LUTs a favor de SIS para los cinco circuitos, que suman un total de 3879 puertas lógicas.

Por último, en las tablas 3 y 4 veremos los resultados del programa para K=5, desglosando el número de 4-LUTs y de 5-LUTs que se requieren para el mapeo de cada circuito según FIPMAP.

Fichero VHD	Nº de puertas	4-LUTs	5-LUTs	Total
SEQ	2273	781	724	1505
ALU4	1289	424	483	907
APEX2	191	60	40	100
5EXP1	71	24	12	36
CLIP	164	44	45	89
MM9B	205	64	21	85
MULT16B	139	63	11	74
S298	1605	393	255	648
TBK	111	34	24	58
MULTIP2	648	146	54	200
TOTAL	6696	2033	1669	3702

Tabla 3: Resultados d FIPMAP para K=5

Además de estas pruebas, orientadas a medir la eficacia del programa de mapeo en cuanto a número de LUTs, se han realizado multitud de pruebas con circuitos diseñados específicamente para casos concretos con el objetivo de buscar fallos de funcionamiento.

El programa se ha testado con circuitos de muy diverso tipo con peculiaridades como cadenas de inversores en cascada, bucles con inversores, circuitos del tamaño de una sola puerta, solo un inversor, etc.

El resultado ha sido satisfactorio en el 100% de las pruebas, y hasta la fecha no se han detectado fallos en su funcionamiento tras haber sido probado con más de 100 circuitos, entre ellos los circuitos *benchmarks* vistos anteriormente.

Destacar que las pruebas no se han limitado a la realización del mapeo y la obtención del fichero JLIF de salida, sino que este fichero se ha introducido de nuevo en el flujo de diseño para continuar con la fase de *place & route* hasta el final. Estas pruebas han sido satisfactorias en el 100% de los casos, sin que se hayan detectado anomalías de funcionamiento hasta el momento de la escritura de esta memoria.

VI. CONCLUSIONES Y RECOMENDACIONES

CONCLUSIONES Y RECOMENDACIONES

Como se ha visto en la sección anterior, los resultados en cuanto a ocupación de área, es decir, en cuanto a número de LUTs son bastante aceptables en comparación con el programa SIS.

Si bien los peores resultados se obtuvieron con el último fichero, "multip2", y en un estudio más extenso lo adecuado sería investigar a fondo el mapeo de este circuito para ver dónde se produce la pérdida de optimización. Este estudio, debido al gran tamaño de estos circuitos, es muy complejo y requeriría sin duda el desarrollo de herramientas específicas que permitan el seguimiento del mapeo, que para este proyecto fin de carrera he realizado de forma manual, con la naturales limitaciones que ello supone.

Sería necesario desarrollar una herramienta visual, que mostrase la evolución de las puertas lógicas y la creación de LUTs en cada una de las fases del algoritmo de mapeo. Además debería mostrar visualmente el resultado final del mapeo a partir del fichero JLIF, indicando los números de las LUTs y los nombres de las señales claramente. Esto último sería necesario para poder comparar los resultados con los de otros programas, obteniendo conclusiones para la optimización del algoritmo de mapeo lógico.

El tiempo de ejecución del programa es muy corto en todos los casos, lo cual indica que este puede ser el planteamiento correcto para el mapeo de circuitos, y que este programa podría ser la base para un sistema de mapeo mucho más complejo introduciendo nuevos mecanismos de optimización y mejorando los que ya se aplican en el código desarrollado. Esto indica que el desarrollo es técnicamente correcto en cuanto a técnicas de programación. Se trata por tanto de mejorar la teoría aplicada de Technology Mapping.

Por otra parte, el programa ha dado muestras de una gran robustez en cuanto a su funcionamiento, ya que en las pruebas realizadas no se han detectado fallos en el mismo.

Destacar también la flexibilidad de funcionamiento que permite el programa al poder elegir entre un mapeo con LUTs de 4 entradas o bien de 5 entradas, y además con un fichero de librería de componentes completamente configurable y modificable por el usuario.

Por último mencionar que hay una vía de investigación en la reconfigurabilidad dinámica aún por explorar. Este programa, por su flexibilidad en la forma en que está programado, podría admitir la incorporación en investigaciones futuras de mecanismos de *hardware swap* si llegan a realizarse.

Esta investigación sería de una complejidad enorme, ya que se trata de desarrollar una forma de dividir los circuitos de entrada en partes "separables", y los estudios que se han desarrollado hasta la fecha y que se pueden usar como base teórica son escasos. En definitiva, este podría quizás ser el tema para un nuevo proyecto fin de carrera completo.

VII. BIBLIOGRAFÍA

BIBLIOGRAFÍA

1. LIBROS Y MANUALES

1. **Technology Mapping for Lookup-Table Based Field-Programmable Gate Arrays**, de Robert J. Francis. Tésis doctoral de la Universidad de Toronto.

2. **Field Programmable System On Chip User's Manual**, FIPSOC Project 1998.

3. **JLIF User Guide**, SIDSA 1998.

4. **VHDL** (2ª edición) de *D.L.Perry*. McGraw-Hill, 1994.

5. **Actel HDL Coding Style Guide**, Actel Corporation 1997.

6. **Principles of CMOS VLSI Design** (2ª edición), de *M. Weste, K. Eshraghian*. Addison-Wesley, 1993.

7. **Foundation Series 2.1i User Guide**. 1999.

8. **SYNOPSYS On-Line Documentation.** 1999.

9. **MPLAB y MPLAB-C On-Line Documentation.** 1999

10. **XESS de Xilinx. On-Line Documentation,** 2000.

2. ARTÍCULOS

1. **A Routability and Performance Driven Technology Mapping Algorithm for LUT Based FPGA Designs**, Chi-Chou Kao & Yen-Tai Lai. National Cheng Kung University, Taiwan 1999.

2. **Multicontext Field-Programmable Gate Arrays**, André DeHon.

3. **Delay Minimal Decomposition of Multiplexers in Technology Mapping,** Shashidhar Thakur, D.F. Wong, Shankar Krishnamoorthy, and P. Moceyunas. *International Workshop on Logic Synthesis,* May 1995.

3. PÁGINAS WEB

1. www.sidsa.es

2. www.xilinx.com

3. www.esi.us.es

VIII. ANEXOS

ANEXO 1: Fichero de librería "fipsoc.gl"

```
# Librería para FIP-MAP

GATE buff       0    O=a;
GATE inv        0    O=!a;

GATE and2       2    O=a*b;
GATE and3       3    O=a*b*c;
GATE and4       4    O=a*b*c*d;
GATE and5       5    O=a*b*c*d*e;

GATE or2        2    O=a+b;
GATE or3        3    O=a+b+c;
GATE or4        4    O=a+b+c+d;
GATE or5        5    O=a+b+c+d+e;

GATE nand2      2    O=!(a*b);
GATE nand3      3    O=!(a*b*c);
GATE nand4      4    O=!(a*b*c*d);
GATE nand5      5    O=!(a*b*c*d*e);

GATE nor2       2    O=!(a+b);
GATE nor3       3    O=!(a+b+c);
GATE nor4       4    O=!(a+b+c+d);
GATE nor5       5    O=!(a+b+c+d+e);

GATE aoi21      3    O=!(a*b+c);
GATE aoi22      4    O=!(a*b+c*d);
GATE oai21      3    O=!((a+b)*c);
GATE oai22      4    O=!((a+b)*(c+d));
```

```
GATE xor2          2       O=a^b;
GATE xnor2         2       O=!(a^b);

# GATE xor2        2       O=a*!b+!a*b;    -- Formas alternativas
# GATE xor2        2       O=!(a*b+!a*!b);
# GATE xnor2       2       O=a*b+!a*!b;
# GATE xnor2       2       O=!(!a*b+a*!b);

GATE mux21         3       O=a*s+b*!s;
GATE mux21         3       O=!(!a*s+!b*!s);

FLIP_FLOPS;

DFF          D NC CLK VDD GND VDD Q        1000010000;
DFFMSS       D D2 CLK   M   S VDD Q        0001010000;
DFFSS        D NC CLK VDD   S VDD Q        1001010000;
DFFM         D D2 CLK   M GND VDD Q        0000010000;
DFFMSR       D D2 CLK   M   R VDD Q        0000010000;
DFFESR       D  E CLK  NC   R VDD Q        1100010000;
```