UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

DEPARTAMENTO DE ELECTRÓNICA Y TELECOMUNICACIÓN

TESIS DOCTORAL

ENTORNO DE AYUDA A LA CONCEPCIÓN, DISEÑO Y SIMULACIÓN DE ARQUITECTURAS CON ALTO GRADO DE PARALELISMO

AURELIO VEGA MARTÍNEZ

Las Palmas de Gran Canaria, Diciembre de 1992

11-1992/93

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

UNIDAD DE TERCER CICLO Y POSTGRADO

Reunido el día de la fecha, el Tribunal nombrado por el Excmo. Sr. Rector Magfco. de esta Universidad, el aspirante expuso esta TESIS DOCTORAL.

Terminada la lectura y contestadas por el Doctorando las objeciones formuladas por los señores jueces del Tribunal, éste calificó dicho trabajo con la nota de APTO CUM LAUDE POR UNANIMIDAD

Las Palmas de G. C., a 21 de Diciembre de 1992

El Presidente: Dr. D. Francisco Rubio Royo,

El Secretario: Dr. D. Roberto Sarmiento Rodríguez,

El Vocal: Dr. D. Javier Uceda Antolín,

El Vocal: Dr. D. Carlos López Barrio,

El Vocal: Dr. D. Emilio López Zapata,

El Doctorando: D. Aurelio Vega Martínez,

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA · UNIDAD DE TERCER CICLO Y POSTGRADO

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

ESCUELA TECNICA SUPERIOR DE INGENIEROS INDUSTRIALES

TESIS DOCTORAL

ENTORNO DE AYUDA A LA CONCEPCION, DISEÑO Y SIMULACION DE ARQUITECTURAS CON ALTO GRADO DE PARALELISMO.

Autor: Aurelio Vega Martínez
Director: Antonio Núñez Ordóñez
Dpto. Electrónica y Telecomunicación
Diciembre-1992

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

DOCTORADO EN INGENIERIA INDUSTRIAL
DEPARTAMENTO DE ELECTRONICA Y TELECOMUNICACION
PROGRAMA DE INGENIERIA ELECTRONICA

Entorno de ayuda a la concepción, diseño y simulación de arquitecturas con alto grado de paralelismo.

Tesis Doctoral presentada por D. Aurelio Vega Martínez
Dirigida por el Dr. D. Antonio Núñez Ordóñez

El Director,

El Doctorando,

A Vega.

Las Palmas de Gran Canaria a 21 de Diciembre de 199_

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

E.T.S. DE INGENIEROS INDUSTRIALES

TESIS DOCTORAL

Entorno de ayuda a la concepción, diseño y simulación de arquitecturas con alto grado de paralelismo.

Autor

D. Aurelio Vega Martínez
Ingeniero Industrial

Director

Dr. Antonio Núñez Ordóñez
Catedrático de Tecnología Electrónica
Universidad de Las Palmas de G.C.

TRIBUNAL

Presidente

Dr. Francisco Rubio Royo
Catedrático de Física Aplicada
Universidad de Las Palmas de G.C.

Vocales

Dr. Carlos López Barrio
Catedrático de Tecnología Electrónica
Universidad Politécnica de Madrid

Dr. Javier Uceda Antolín
Catedrático de Tecnología Electrónica
Universidad Politécnica de Madrid

Dr. Emilio López Zapata
Catedrático de Arquitectura y Tecnología de Computadores
Universidad de Málaga

Dr. Roberto Sarmiento Rodríguez
Titular Interino de Tecnología Electrónica
Universidad de Las Palmas de G.C.

A Macu

RESUMEN

En los último años, se ha realizado a nivel internacional un esfuerzo importante conducente a la definición de nuevos estándares en lenguajes de descripción hardware, síntesis, simulación, entornos CAD de diseño, etc. Cada vez tenemos maquinas más potentes y herramientas más versátiles a la hora de trabajar. Las mejoras han sido grandes, pero aún no se ha logrado eliminar una de las piezas del engranaje de la maquinaria del mundo del diseño electrónico: el diseñador.

Según disponemos de herramientas de trabajo más sofisticadas, los diseños que se pretende realizar son a la vez más complejos. Las herramientas de diseño disponibles nos eliminan las tareas más tediosas del proceso, permitiendo la concentración en la problemática planteada a muy alto nivel, aunque no todas las herramientas nos ayudan de forma óptima a desarrollar las cualidades innatas necesarias para el diseño.

El objetivo del presente trabajo es el desarrollo de un conjunto de herramientas que ayuden al diseñador durante las fases de concepción, diseño y simulación de arquitecturas digitales VLSI, con especial énfasis en las arquitecturas con alto grado de paralelismo.

Se presenta un nuevo lenguaje de descripción hardware que sirve de nexo de unión entre el diseñador y la plataforma CAD sobre la que se trabaje. La descripción de las nuevas herramientas desarrolladas, así como la metodología de trabajo a emplear constituyen el cuerpo de esta tesis. Además se realiza una evaluación comparativa de la metodología propuesta, fundamentada en el análisis de los resultados obtenidos con la aplicación de la metodología propuesta sobre varias arquitecturas de referencia.

Se finaliza con un conjunto de conclusiones sobre el grueso del trabajo realizado, así como con una guía de campos afines al de la simulación, sobre los cuales podemos extender el horizonte de todo lo expuesto.

AGRADECIMIENTOS

El trabajo realizado en esta tesis no es únicamente el resultado de un esfuerzo personal. Quiero expresar mi sincero agradecimiento a todas aquellas personas que de muy diversas maneras me han ayudado y alentado para hacer realidad el trabajo que aquí se presenta.

A mi tutor, Antonio Núñez Ordóñez por su impecable labor de dirección, consejo y revisión del trabajo, sin cuya ayuda me hubiese sido imposible llegar hasta el final.

A Roberto Sarmiento y Pedro P. Carballo por su incondicional amistad y apoyo durante todos los años que hemos estado trabajando juntos.

A Jorge Monagas, Roberto Esper-Chain y Juan Carlos por su buen hacer y por las muchas horas dedicadas a la realización y mejora de los programas de esta tesis. A Carlos, Juan Manuel y tantos otros del Departamento de Electrónica y Telecomunicación, que de una forma u otra me han ayudado en éste y en otros proyectos.

Al Departamento de Ingeniería Electrónica de la E.T.S.I.T. de Madrid en donde me recibieron durante los primero años de mi formación en el campo de la microelectrónica, y donde se comenzó a concebir esta tesis. Y en especial a Carlos López Barrio y Juan Carlos López.

A mis padres por su incondicional apoyo y aliento.

Por último quiero agraceder a Macu su comprensión por todas las horas que he tenido que robarle para dedicarme al trabajo.

A todos,

Gracias.

INDICE

INDICE DE FIGURAS

INDICE DE TABLAS

Capítulo 1.

Introducción.

1.1 Planteamiento del problema, necesidad y utilidad de esta investigación.

Los entornos de diseño CAD nos permiten abordar todas las etapas del diseño electrónico, desde la especificación en alto nivel de un sistema complejo, pasando por el diseño de circuitos impresos o circuitos integrados hasta las etapas finales de cableado o diseño mecánico.

La tendencia consolidada desde hace ya algunos años es ir hacia entornos de diseño integrados en los que poder abordar las distintas etapas de diseño sin necesidad de acudir a herramientas inconexas. Los entornos de diseño nos permiten afrontar las distintas actividades a realizar sin tener que cambiar de forma de trabajo al pasar de una herramienta a otra.

Una de las piezas claves de cualquier sistema de diseño es la capacidad de simulación. La especificación y simulación de sistemas complejos ha evolucionado hacia la utilización de lenguajes de descripción hardware. Los distintos sistemas de descripción hardware han conducido a la definición de estándares como el VHDL.

Todos los sistemas comerciales se basan para la descripción del sistema que pretenden simular en los siguientes puntos:

1) Descripción estructural del sistema mediante la utilización de esquemáticos.

2) Descripción estructural y de comportamiento mediante el empleo de diversos lenguajes de descripción hardware (HDL).

3) Sistemas mixtos en los que una parte de la lógica está descrita mediante el procedimiento clásico de esquemáticos en los que alguno de los componentes están descritos mediante algún lenguaje HDL.

Sobre este punto hay que considerar lo siguiente:

1) En las arquitecturas VLSI con alto grado de regularidad, la descripción de estos sistemas utilizando los lenguajes HDL genéricos no son del todo eficientes, ya que normalmente éstos no poseen comandos específicos que faciliten la labor de su descripción. De igual forma las herramientas de captura de esquemáticos, aunque dispogan de comandos para repetir o crear arrays, no están especialmente pensadas para trabajar con este tipo de arquitecturas.

2) En cualquiera de las opciones de entrada de datos al simulador, una vez generada la información sobre conectividad, se procede a su simulación empleando alguno de los lenguajes clásicos de especificación de vectores. Los resultados de simulación son normalmente comprobados utilizando los típicos diagramas de estados lógicos. La verificación de los resultados de la simulación puede resultar tediosa cuando estamos verificando gran número de señales.

3) Existen algunos sistemas CAD, que permiten la descripción del sistema a simular mediante una herramienta de descripción de bloques parecida a una herramienta de captura de esquemáticos. En éstos, conjuntamente con la descripción estructural se añaden bloques de control que permiten visualizar valores de variables internas. De forma que los resultados de simulación se reflejan en el propio esquemático, dentro de bloques de control, como valores numéricos que visualizan el valor de alguna de las variables internas. De esta forma se eliman los diagramas de estados lógicos en favor de esta otra forma de visualización.

En esta tesis se ha desarrollado un lenguaje denominado **LDAP** (**L**enguaje de **D**escripción de **A**rquitecturas **P**aralelas), que no pretende ser un lenguaje completo de descripción de hardware, sino un mecanismo de descripción rápida de las interconexiones que nos podemos encontrar en arquitecturas con alto grado de paralelismo. Las descripciones realizadas con este lenguaje se transformarán en descripciones estructurales en lenguajes ampliamente utilizados como el VHDL o Verilog. Será labor del diseñador completar la descripción de las unidades básicas de este lenguaje (células) utilizando el lenguaje de descripción hardware seleccionado.

Aquí se propone también que el interfase del simulador con el usuario, sea distinto de los habituales. En los sistemas con alto grado de paralelismo no es tan importante ver en un diagrama la evolución de una determinada señal durante un determinado número de ciclos, sino que lo que nos interesa es cómo evoluciona el conjunto de los datos a golpes de ciclos de reloj.

El objetivo es tratar cualquier simulación como un sistema que se pretenda monitorizar y controlar, teniendo claramente a la vista como evolucionan sus distintas variables. En procesos de telemando y telecontrol industrial lo más importante para el operador es saber en un determinado estado de tiempo cuál es el valor de las distintas variables, y comprobar si todas ellas se encuentran dentro de los márgenes normales de trabajo.

Esta idea tomada de las herramientas de ingeniería de sistemas industriales está en la base de esta tésis, al expandir su aplicación a arquitecturas con alto grado de paralelismo. Al aumentar la observabilidad de la arquitectura se hace mucho más fácil, no sólo su comprensión sino también su concepción, diseño y simulación.

Este tipo de observaciones no eliminan a las clásicas de formas de onda, sino que complementan su eficacia. La eficacia de este tipo de visualización o vista de una simulación será mayor cuanto mayor sea el grado de paralelismo de los distintos procesos.

1.2 Objetivo de la Tesis y aportaciones.

El objetivo de esta Tesis es el desarrollo de un conjunto de herramientas denominado **EASAP** (Entorno de Ayuda a la concepción, diseño y Simulación de Arquitecturas con alto grado de Paralelismo).

Como iremos viendo a lo largo de la exposición, los distintos tipos de arquitecturas que presentan un alto grado de regularidad, poseen unas características comunes que no son siempre adecuadamente tratadas en los entornos CAD comerciales.

Esta tésis pretende agilizar los primeros estadios de concepción del diseño de este tipo de arquitecturas, facilitando el camino a la simulación y realización física del sistema mediante herramientas estándar. Nuestro entorno EASAP es una ayuda al diseñador en los puntos comentados anteriormente por las siguientes razones:

Ayuda a la concepción:

Podemos realizar una descripción estructural de una arquitectura paralela y visualizar ésta gráficamente de forma muy rápida, lo que nos facilita la verificación y corrección de errores.

El tener de forma rápida una imagen de la arquitectura descrita acelera las etapas iniciales del diseño.

En este tipo de arquitecturas, la capacidad de ver evolucionar los flujos de datos sobre una representación gráfica de la arquitectura descrita, da al diseñador más y mejor información que la suministrada por un diagrama lógico o tablas de estados.

La característica de ser un entorno abierto nos permite la conexión de programas que implementen algoritmos generadores de arquitecturas a partir de descripciones algorítmicas. Esto nos permitirá concentrarnos en el algoritmo utilizado.

La posibilidad de mostrar simultáneamente arquitecturas y flujos de datos nos da una herramienta de formación sumamente potente.

Ayuda al diseño:

Una vez realizada la descripción estructural de una arquitectura objeto de estudio en nuestro entorno, se puede trasvasar directamente al entorno CAD en el cual se continuará con las labores de realización física, evitando tareas de muy bajo nivel o muy monótonas.

Podemos incorporar nuevas funciones de evaluación tecnológica de las arquitecturas VLSI descritas, con lo que al visualizar éstas, podemos tener además una estimación del área, potencia, etc., lo que supone tener información valiosa sobre variables a tener en cuenta a la hora de optar por arquitecturas alternativas.

Ayuda a la simulación:

Al incorporar la descripción estructural de nuestra arquitectura a la base de datos del entorno CAD con el que trabajemos, podemos realizar su simulación con el simulador que se encuentre soportado.

La herramienta de visualización gráfica propuesta, puede ser conectada a distintos entornos facilitando las tareas de simulación y comprensión de los resultados.

1.3 Ordenación de esta memoria.

La presente memoria está dividida en cinco capítulos, incluido este primero de introducción.

El capítulo segundo nos presenta una visión general de tres campos bien diferenciados que tienen relación con esta tesis. En primer lugar se realiza una revisión del tipo de arquitecturas que van a ser estudiadas. Se analizan en segundo lugar los lenguajes de descripción hardware y se concluye con un estudio sobre entornos CAD de ayuda al diseño y simulación, todo ello en lo referente al soporte que dan a la concepción, diseño y simulación de arquitecturas con alto grado de paralelismo, objeto central de esta tesis.

El capítulo tercero es una introducción al entorno de trabajo propuesto, EASAP. Se explica de una forma somera los conceptos generales sobre los que se sustenta el trabajo, se describen las distintas herramientas y lenguajes desarrollados, así como los métodos de trabajo que se pueden emplear. Se termina con unas notas sobre la implementación final realizada.

El capítulo cuarto está dedicado a unos ejemplos de trabajo con EASAP, y su implementación.

En el capítulo quinto encontraremos una descripción detallada de los lenguajes y herramientas de EASAP.

Se finaliza con la presentación de las conclusiones obtenidas en el presente trabajo. Se comentan las aportaciones realizadas y las líneas de trabajo que se están siguiendo para la ampliación del ámbito de aplicación de esta tesis.

Capítulo 2.

Visión general de herramientas y lenguajes para arquitecturas paralelas.

Tal como se ha explicado en el Capítulo 1 esta tesis abarca tres grandes campos: arquitecturas VLSI con alto grado de paralelismo, lenguajes de descripción hardware y entornos CAD.

El primero de ellos está relacionado con esta tesis sólo por el hecho de ser el principal campo de aplicación de los lenguajes y herramientas que se desarrollan. Como tal campo de aplicación fija el grado de ayuda y soporte, y por lo tanto, los prerequisitos y especificaciones, que se desea obtener de las herramientas. En este capítulo se hace una revisión somera de las arquitecturas paralelas para pasar a estudiar un poco más en detalle las arquitecturas sistólicas, con el doble objetivo de reflexionar sobre las necesidades de soporte estructural y funcional de estas arquitecturas, y de hacer la exposición de la terminología y conceptos usados en esta tesis lo más autocontenida posible, facilitando así su comprensión.

El segundo y tercer campo citados, lenguajes de descripción hardware y entornos CAD, corresponden al núcleo de las aportaciones de la tesis y su revisión es necesaria por dos motivos. En primer lugar porque justamente el análisis de las limitaciones de los HDL y entornos CAD disponibles en el diseño VLSI para las arquitecturas indicadas, determinó en su momento el planteamiento de esta tesis. En segundo lugar porque las aportaciones de los lenguajes y entorno creado por EASAP deben evaluarse por sus prestaciones en comparación con las ayudas que actualmente ofrece la tecnología de HDL y CAD disponible.

2.1 Arquitecturas VLSI con alto grado de paralelismo.

Existen gran cantidad de estructuras o arquitecturas VLSI que de una forma u otra utilizan el procesamiento paralelo como herramienta habitual de trabajo. Pensemos por ejemplo en los sumadores, multiplicadores, ALUs o Data Paths en los que tenemos estructuras replicadas que operan en paralelo sobre los diferentes bit de las palabras de entrada, para llegar lo antes posible a la obtención del resultado buscado. De igual forma es evidente el paralelismo innato que podemos encontrar en una FIFO o en un banco de memoria.

En otras arquitecturas VLSI, como es el caso de las arquitecturas sistólicas, este paralelismo es llevado más allá, ya que es la misma esencia de su forma de trabajo. Por este motivo, tomaremos este tipo de arquitecturas como modelo de referencia, de donde poder extraer todos aquellos conceptos básicos que den soporte a las herramientas que se van a presentar, y que se puedan exportar o trasladar a otras estructuras o arquitecturas VLSI.

Para poder captar estos conceptos básicos, vamos a presentar en primer lugar varias de las clasificaciones clásicas de arquitecturas paralelas para tener una visión de conjunto de la

problemática que se plantea, para pasar a continuación a un estudio más detallado de las arquitecturas sistólicas.

2.1.1 Clasificación de las arquitecturas paralelas.

A lo largo de la historia reciente se han desarrollado muy diversos tipos de arquitecturas paralelas para cubrir las necesidades crecientes de cálculo. Los computadores han evolucionado en velocidad de cálculo y en capacidad de almacenamiento, según los problemas que se pretendían solucionar aumentaban en complejidad y en volumen de información a tratar.

Es aceptado por todos que el **procesamiento secuencial** que se utiliza en los computadores individuales ha de dejar paso al **procesamiento paralelo** en el que para resolver un mismo problema recurrimos al empleo de diversas unidades de computación trabajando conjuntamente para la obtención del fin común.

No existe una única clasificación que englobe los muy diversos tipos de computadores paralelos presentados hasta la fecha. Una misma arquitectura puede presentar características que le permitan pertenecer a distintas clasificaciones simultáneamente, mientras que otras pueden difícilmente encajar en alguna de las propuestas. De todas formas existen algunas clasificaciones más o menos generalmente aceptadas como la de Flynn [Fly72] en la que se establecen cuatro grupos de computadores en función del tipo de cadenas de datos y secuencia de instrucciones:

SISD (Single Instruction, Single Data streams)

En este tipo de computadores, existe una única cadena de instrucciones que se ejecuta secuencialmente sobre una única cadena de datos. Los computadores secuenciales convencionales son los pertenecientes a este grupo.

SIMD (Single Instruction, Multiple Data streams)

En las maquinas SIMD, una misma instrucción se ejecuta sobre distintas unidades de procesamiento que trabajan cada una sobre distintas cadenas de datos.

MISD (Multiple Instruction, Single Data streams)

En este caso tendríamos distintas unidades de procesamiento ejecutando distintas secuencias de instrucciones sobre una misma cadena de datos. No existe ningún tipo de computador real perteneciente a esta clasificación.

MIMD (Multiple Instruction, Multiple Data streams)

Cada una las unidades de procesamiento ejecutarían distintas secuencias de instrucciones sobre distintas cadenas de datos.

Otra clasificación más interesante para el objeto de esta tesis es la que se fundamenta en el

tipo de estructura del computador paralelo [HwBr88]. Los grupos que podemos encontrar según este criterio son tres:

Computadores Vectoriales:

Estos segmentan el flujo de instrucciones durante el procesamiento, permitiendo que los distintos segmentos se ejecuten solapadamente.

Sistemas Multiprocesador:

Estos sistemas están formados por dos o más procesadores que comparten recursos comunes como son bancos de memoria o dispositivos de E/S, además de disponer de cualquier tipo de recursos locales. Las comunicaciones entre los distintos procesadores se pueden realizarse a través de distintos mecanismos como memorias compartidas.

Computadores Matriciales:

Están formados por múltiples unidades de procesamiento elementales que operan síncronamente sobre distintos datos. Necesitan de la existencia de una unidad de control o **host** que se encargue de la supervisión del proceso.

Al grupo de computadores matriciales pertenecen las distintas arquitecturas sistólicas. Nuestro interés se va a centrar en este tipo de arquitecturas ya que además de su innegable interés en el campo del procesamiento digital de las señal, presentan la característica de ser implementables a nivel VLSI.

2.1.2 Arquitecturas sistólicas.

El término sistólico significa contracción. Está derivado del nombre *sístole* el cual se refiere a la contracción de la sangre bombeada desde el corazón a través del sistema arterial. De forma similar a un sistema biológico un sistema sistólico VLSI bombea rítmicamente una cadena de datos a través de un array de células.

Existen diversas definiciones sobre lo que significa una arquitectura sistólica, si bien todas ellas coinciden en sus aspectos esenciales. La utilización de tecnologías de diseño VLSI para la realización de circuitos integrados posibilita el desarrollo de potentes sistemas de computación en un único circuito integrado formado por elementos de procesamiento conectados de forma regular.

El concepto de arquitecturas sistólicas fue desarrollado en la Universidad de Carnegie-Mellon [KunH78, KunH80/3, KunH82/1]. Definiciones formales pueden encontrarse en la tesis doctoral "Area Efficient VLSI Computations" por Charles E. Leiserson [Lei81/1] y también en [Lei83, Lei81/2]. Una descripción general de los términos empleados corrientemente en el trabajo con este tipo de arquitecturas, puede ser encontrada en la Tabla II.1. Aquellos sistemas que combinan las características de *multi-procesamiento* y *segmentación encauzada*, son referenciados como arquitecturas sistólicas.

Por **multi-procesamiento** se entiende un conjunto de computaciones realizado simultáneamente en diferentes células de procesamiento.

Se pueden reconocer dos tipos de **segmentación encauzada** (*pipelining*) en las arquitecturas sistólicas

* **Segmentación en los datos** (*data pipelining*). Reduce los requerimientos de ancho de banda de entrada salida del entorno por el uso múltiple de cada valor de la cadena de datos.

* **Segmentación en los cálculos** (*computational pipelining*). Está indicada por la propiedad de que cada computación se realiza ejecutando varias operaciones en diferentes células de procesamiento, por ejemplo, una célula de procesamiento produce una salida parcial la cual entra en la célula vecina. La ejecución de las operaciones realizadas en una célula puede también estar segmentada, por lo que la segmentación del cálculo se realiza en dos niveles.

Una **arquitectura sistólica** es una estructura formada por células de procesamiento básicas interconectadas cumpliendo las siguientes características:

* las células de procesamiento son pequeñas, de forma que cada una de ellas realiza unas pocas operaciones básicas, como operaciones aritméticas.

* las células básicas no realizan necesariamente las mismas operaciones, pudiendo ser funcionalmente diferentes.

* las geometrías asociadas a cada célula básica pueden ser rectangulares o hexagonales.

* debe existir un interfase externo con el host.

* las comunicaciones entre las células de procesamiento siempre son locales.

* las comunicaciones entre células de procesamiento están sincronizadas mediante la utilización de una señal de reloj global.

Las **arquitecturas semi-sistólicas** poseen todas las características de las sistólicas con la excepción de que las comunicaciones entre las células no tienen por qué utilizar reloj, por ejemplo, también pueden tener *broadcasting* entre el host y las células de procesamiento y/o propagación entre las células.

Ya que en una arquitectura puramente sistólica no existe propagación ni *broadcasting*, el período de la señal de reloj global es independiente del tamaño del array y depende del tiempo de retraso de una célula de procesamiento básica, sí no consideramos el retraso inherente a las interconexiones entre células. De cualquier forma en el caso de grandes arrays el *skew* del reloj llega a ser un problema. En [Fis85/2, Lei81/2, Kung88] se introducen modelos de sincronización y basándose en esos modelos apropiados se proponen métodos para la sincronización de grandes arrays. Una aproximación alternativa a la sincronización mediante una señal de reloj global es la *auto-temporización* conduciendo a array de frente de onda asíncronos (*wavefront arrays*) [KunS87, Wei81].

Tabla II.1 Definiciones de términos utilizados con arquitecturas sistólicas.

Célula:	Una célula es una unidad hardware funcional. Una descripción de una célula consiste en una vista externa y en la especificación de una vista interna.
Puerto:	Un puerto es un pin (conector) o un conjunto de pines, mediante los cuales una célula se comunica con su entorno.
Vista externa de una célula:	Se denomina vista externa de una célula a la parte de la descripción de una célula, que especifica los puertos de la célula por sus tipos y por su localización en la periferia de la célula.
Vista interna de una célula:	Se denomina vista interna de un célula la parte de la descripción de ésta, que especifica la característica y/o la estructura de la célula.
Array:	Un array es una red de células funcionalmente idénticas interconectadas entre sí.
Célula básica de un array:	Se denomina célula básica de un array a la unidad más pequeña de procesamiento repetidamente usada para componer el array.
Floor plan:	Se entiende por "floor plan" a la representación del diseño de un CI, la cual abstrae el tamaño físico de varias células y especifica únicamente las relaciones topológicas entre ellas.
Host:	Un host es el ordenador primario de una red de computación. El host se usa para preparar programas o datos para el uso de otro ordenador u otro sistema de procesamiento de datos.
Comunicaciones globales:	Dado un sistema hardware de varias células. El camino de comunicaciones se denomina "comunicación global" si existe una interconexión directa entre el host y cada una de las células o viceversa.
Comunicaciones locales:	Dado un sistema hardware de varias células. El camino de comunicaciones se denomina "comunicación local" si las células se comunican sólo con sus vecinas.
Difusión:	Si un dato es transferido desde una fuente directamente a varias células destino, entonces esa comunicación se llama de difusión (broadcasting) y el dato transferido se referencia como difundido.
Propagación:	Dado un dato el cual se mueve a través de un array de células de procesamiento. En la célula su valor puede ser modificado de acuerdo con la operación realizada en la célula o no. Si no hay registros localizados en los respectivos caminos de datos, entonces el flujo de datos se llama de "propagación".
Cadena de datos:	Una secuencia de valores en donde cada valor se referencia por un subíndice identificador, por ejemplo a(1), a(2), ..., a(n), se llama una "cadena de datos". Una cadena de datos cuyos valores se introducen desde el host, se llama "cadena de datos de entrada". Similarmente una cadena de datos, cuyos valores son generados por una arquitectura sistólica y son leídos por el host, se llama "cadena de datos resultantes".
Variable de una cadena de datos:	Considerando un algoritmo, el identificador que nombra a una cadena de datos se llama "variable de la cadena de datos".
Variable de índice:	Dado un algoritmo escrito como un programa de bucles anidados o como un ecuación recurrente. Las variables que controlan el bucle o las iteraciones recurrentes se llaman "variables de índice".
Skew:	Desfase en las señales de reloj.

Para el objeto de esta tesis es de especial interés analizar la geometría y topología de la red, así como su conectividad y flujo de datos.

Dependiendo de la forma del contorno de las células básicas, rectangulares o hexagonales, el *floor plan* de las arquitecturas sistólicas podríamos clasificarlo como:

* lineal.
* rectangular y cuadrangular.
* triangular.
* hexagonal.

Las cadenas de datos pueden viajar a diferentes velocidades a través del array. La velocidad

de las cadenas de datos está determinada por el número de registros localizados en el camino de datos (*data path*) entre dos células vecinas [Lem89]. En el ejemplo de la Fig. 2.1 podemos ver que entre dos células en el camino de datos de la cadena de datos **X** encontramos dos registros, y como en el camino de datos de la cadena de datos **Y** hay sólo un registro, un valor de la cadena de datos **X** necesita el doble de tiempo de un valor de la cadena de datos **Y** para viajar a través del array.

Considerando las tecnologías disponibles hoy en día, la mayoría de las arquitecturas sistólicas de dos dimensiones pueden ser realizadas en un único circuito integrado. En un array de una dimensión la célula básica puede recibir datos desde dos células vecinas como se ilustra en la Fig. 2.2. En arrays de dos dimensiones compuestos de células básicas rectangulares las células tienen un máximo de cuatro posibilidades para comunicarse con sus vecinas (Fig. 2.3 y Fig. 2.4). En un array compuesto de células hexagonales hay dos posibilidades de comunicaciones rectangulares y cuatro diagonales (Fig. 2.5).

Tal como hemos visto en los ejemplos anteriores, en un array una cadena de datos que entra en una célula por uno de sus lados, sale siempre por el lado opuesto. Debido a ésto, la vista externa de una célula básica puede ser generada a partir de la información sobre qué lado recibe datos desde sus vecinas (Fig. 2.6).

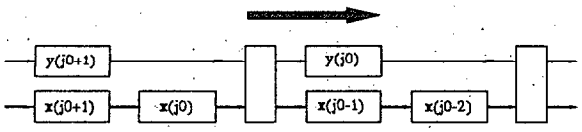

Fig. 2.1 Velocidad de las cadenas de datos.

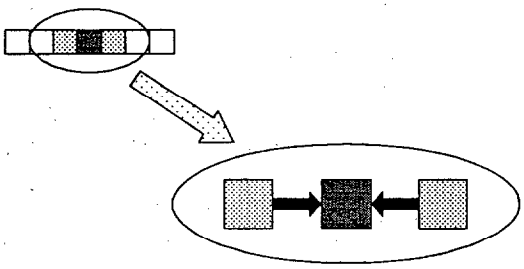

Fig. 2.2 Array lineal.

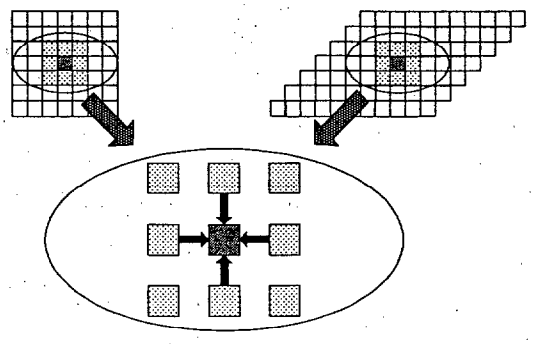

Fig. 2.3 Arrays rectangulares y cuadrangulares.

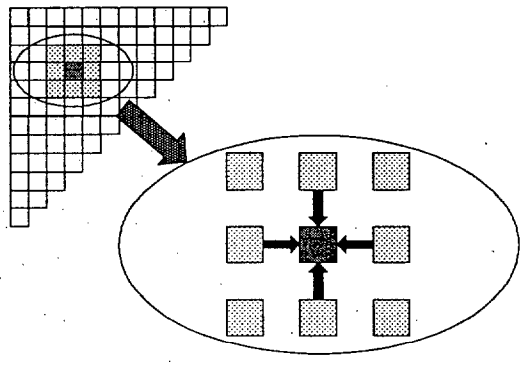

Fig. 2.4 Array triangular.

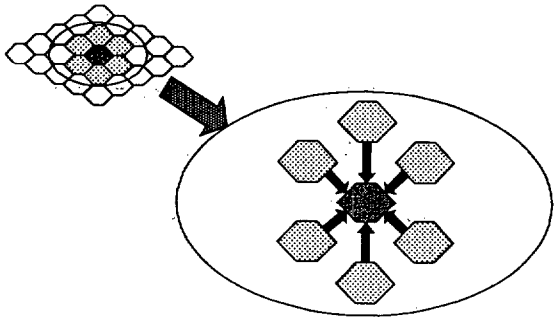

Fig. 2.5 Array hexagonal.

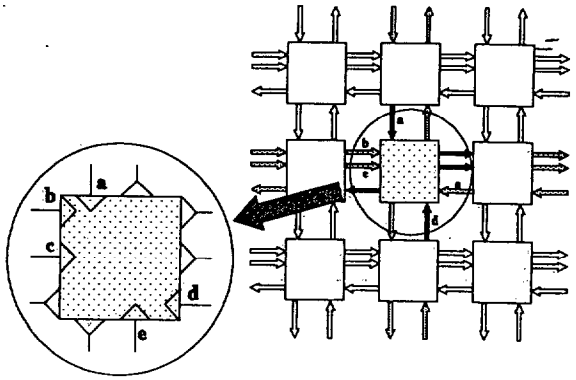

Fig. 2.6 Determinación de la vista externa de una célula.

Un concepto importante a la hora de trabajar con arquitecturas sistólicas es el rendimiento que podemos obtener de ella. Este rendimiento de una arquitectura sistólica o semi-sistólica se mide por una serie de criterios de calidad como el tiempo de inicialización, tiempo de latencia, razón de *throughput* y tiempo de ejecución.

Tiempo de inicialización.

El tiempo de inicialización de una cadena de datos resultado es el tiempo después del cual, todos los valores de cadenas de datos de entrada necesarios para computar el primer valor de la cadena de datos resultado han sido introducidos por el host en el array.

Tiempo de latencia.

El tiempo de latencia de una cadena de datos resultado especifica la medida de tiempo desde el final de la fase de inicialización hasta el momento en que el primer valor de la cadena de datos resultado está disponible para el host.

Razón de throughput.

El *throughput* de una cadena de datos resultado especifica la frecuencia con la cual los datos van llegando al host después de que el primer resultado ha sido recibido.

Tiempo de ejecución t_{exe}.

El cálculo del tiempo de ejecución de un algoritmo se realiza en función de si la arquitectura es sistólica o semi-sistólica.

arquitecturas sistólicas:

$$t_{exe} = \frac{\max\{th\} - \min\{th\} + 1}{\min\{reg(dato)\}}$$

arquitecturas semi-sistólicas

$$t_{exe} = \max\{th\} - \min\{th\} + 1$$

en donde max{th} indica el paso de tiempo de ejecución de la última ejecución y min{th} el paso de tiempo de ejecución de la primera computación. En el caso de arquitecturas sistólicas min{reg{datos}} indica el número más pequeño de registros localizado en el camino de comunicación entre dos células vecinas.

El tiempo de ejecución t_{exe} es determinado por la diferencia incrementada en uno del paso de tiempo en que la última computación ha sido realizada y el paso de tiempo de la primera computación. Si el menor número de registros entre dos células vecinas es mayor que uno, entonces el valor ha de ser dividido por el número menor de los registros.

2.2 Lenguajes de descripción hardware.

Antes de exponer cuales son las principales características de los lenguajes de descripción hardware, hay que explicar una serie de conceptos generales sobre la jerarquía de los sistemas digitales con los que vamos a trabajar.

2.2.1 La jerarquía de los diseños.

Desde el punto de vista de los diseñadores hardware, la jerarquía de un diseño puede ser expresada en términos de niveles de abstracción y en términos de dominios [Arms83, Arms88].

Podemos representar los distintos **niveles de abstracción** de un sistema digital como una pirámide cuya base está constituida por el nivel más bajo de detalle de un circuito que es el **layout**. Este nivel nos está indicando la formas geométricas básicas de un circuito, como pueden ser las áreas de difusión, polysilicio o metal. Según vamos ascendiendo a nivel superiores nos encontramos con representaciones cada vez con menor grado de detalle. Así, por encima del nivel de layout nos vamos encontrando, el nivel de **circuito**, el nivel de **puerta**, el nivel de **registro**, nivel de **chip**, y por último el nivel **PMS** (Processor Memory Switch). En la Fig. 2.7 podemos ver representados estos seis niveles con ejemplos de circuitos representados para cada uno de ellos.

Los dominios posibles dentro de la jerarquía de un diseño son dos: **el dominio estructural** y el **dominio de comportamiento**.

Dominio estructural: En este dominio los componentes están descritos en términos de interconexiones de primitivas o componentes de un nivel inferior.

Dominio de comportamiento: En este dominio, los componentes están descritos mediante un proceso que responde a sus entradas/salidas.

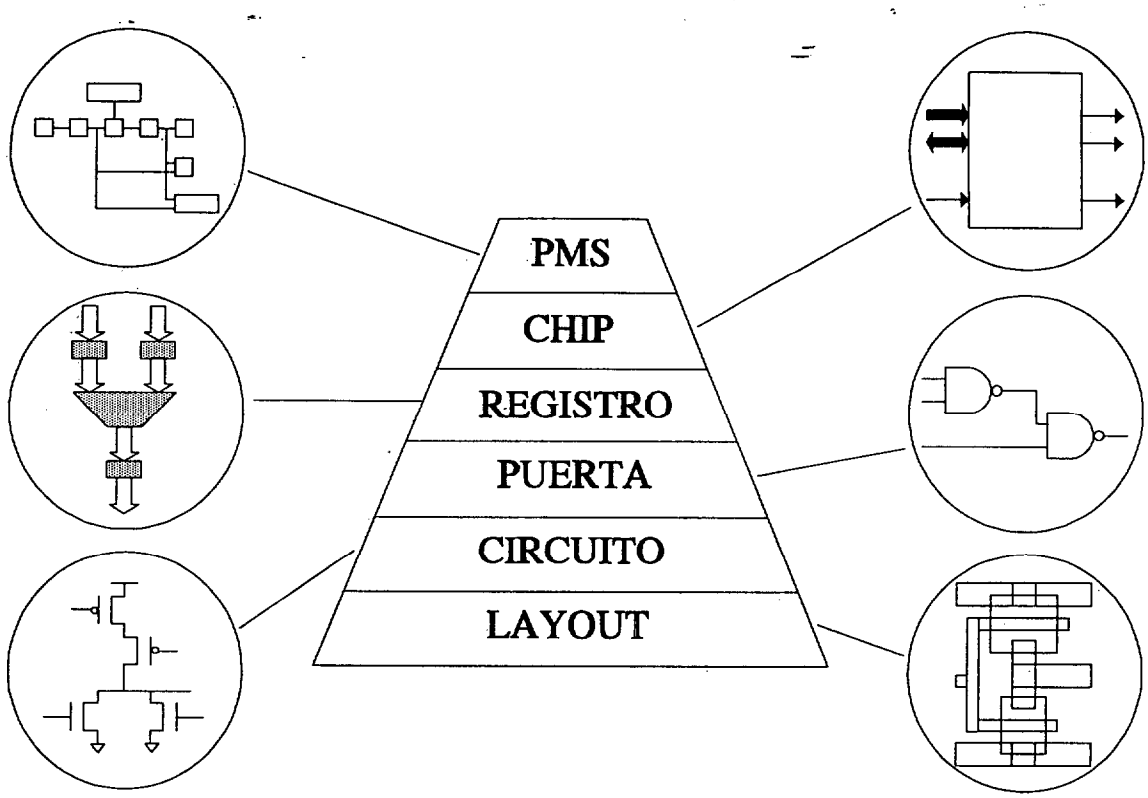

Fig. 2.7 Niveles de abstracción en sistemas digitales.

Tabla II.2 Jerarquías de un diseño.

Primitivas Estructurales	NIVEL	Representación de Comportamiento
Memorias, CPUs, buses	PMS	Especificaciones
RAMs, ROMs, Microprocesadores	CHIP	Algoritmos, micro-operaciones, respuestas de E/S
Registros, Contadores, ALUS	REGISTRO	Tablas de estados, tablas de verdad, micro-operaciones
Puertas, flip-flops	PUERTA	Ecuaciones booleanas
Transistores	CIRCUITO	Ecuaciones diferenciales
Máscaras	LAYOUT	No tiene representación

El la Tabla II.2 se muestra la jerarquía en términos de niveles de abstracción y en términos de dominios.

Descomposición estructural de un diseño.

El hecho de que un diseño se pueda representar en forma estructural implica que vamos a tener que recurrir a un proceso de descomposición de éste. Al describir un diseño en términos de interconexiones entre primitivas, hay que explicar que una primitiva de un nivel, puede estar presentada por primitivas del nivel inferior. Estas relaciones entre diferentes niveles de abstracción la tenemos representada en la Fig. 2.8.

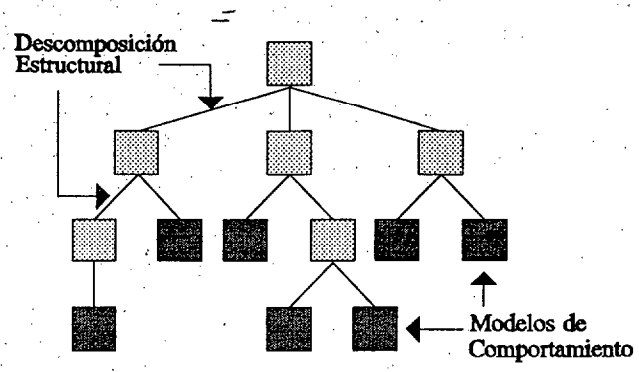

Fig. 2.8 Descomposición estructural de un diseño.

El modelo de comportamiento es un modelo de primitiva en el que las operaciones del modelo están especificadas por medio de procedimientos, en lugar de estarlo en términos de componentes. Tal como se representa en la Fig. 2.8 en un mismo diseño podemos encontrar componentes que no están descritos en el mismo nivel de abstracción, con lo que podemos tener simultáneamente partes de un diseño descritas a nivel de puertas, nivel de circuito o nivel de registro. Por este motivo es necesario el empleo de *simulación multinivel*, en la que podamos trabajar con componentes descritos en diversos niveles de abstracción.

Tal como se desprende del análisis anterior, los lenguajes HDL cumplen con su objetivo de descripción de circuitos en distintos niveles de abstracción. El diseñador recurre a los múltiples mecanismos a sus disposición para la descripción de su sistema, pero esta descripción ha de ser siempre textual. Cuando dentro de un entorno CAD describimos sistemas jerarquizados, podemos optar por realizar esta descripción utilizando exclusivamente un HDL o bien recurrir a la creación de un esquemático con la información estructural del diseño, dejando la descripción funcional o de comportamiento de los distintos módulos al lenguaje HDL.

* En el primer caso, la descripción completa del diseño utilizando un lenguaje HDL no lleva implícita una clara descripción visual del diseño. Es muy difícil algunas veces leer el fichero HDL y hacernos rápidamente una composición mental de cómo están relacionados los distintos elementos de la descripción estructural contenida.

* En el segundo caso, entramos al diseño a través de una hoja de esquemáticos en donde la información estructural del diseño está reflejada en la forma clásica de bloques (componentes) e interconexiones entre ellos. El modelo de simulación de cada bloque situado en el diseño estará descrito mediante una descripción HDL.

Cuando un diseñador se enfrenta al reto de describir el comportamiento de un sistema, enseguida surge la necesidad o la conveniencia de particionar el problema en bloques o unidades de comportamiento claramente diferenciados. De igual forma, cuando abordamos el diseño de un ASIC pensamos en él como una unidad o bloque con entidad propia. El ASIC lo vamos a tratar como una unidad dentro de un sistema más complejo. Pero cuando tenemos que describir la funcionalidad de éste a partir de unos requerimientos de señales de entrada y salida, comenzamos a pensar en él no como un elemento uniforme, sino como un conjunto de unidades con funcionalidad claramente diferenciada (memorias, data path, unidades de control, etc.). De esta manera cada uno de estos elementos podrán ser descritos de forma más

eficiente. Es decir hasta el momento hemos seguido una metodología de diseño *top-down.*

En el caso de arquitecturas con alto grado de paralelismo, como pueden ser las arquitecturas sistólicas, su descripción puede realizarse de dos formas distintas: la primera considerando toda la arquitectura como una unidad, y la segunda considerando las células básicas como las unidades de diseño.

* En el caso de considerar toda la arquitectura como una unidad, el trabajo de descripción suele resultar sencillo debido a que este tipo de arquitectura implementan algoritmos de poca complejidad como convoluciones, transformadas, etc., fácilmente descritas utilizando un HDL.

* En el segundo caso podemos considerar que las unidades más importante del diseño son las distintas células que lo componen. Aquí encontramos una clara diferencia entre la forma de describir la funcionalidad, o comportamiento de nuestro elemento base que es la célula, y la forma de descripción de la arquitectura formada a partir de ésta. Debido a la particularidad de ser arquitecturas regulares, es mucho más eficiente realizar su descripción a nivel estructural, dejando la descripción de la célula en forma funcional o de comportamiento.

2.2.2 Características básicas de los HDL.

Según ha ido aumentado la complejidad de los diseños VLSI en función del aumento del número de componentes, se ha requerido un proceso de diseño mejor estructurado. En respuesta a esta necesidad se ha realizado durante los últimos años un esfuerzo importante en el desarrollo de lenguajes de descripción hardware (HDL: Hardware Description Language) que permitan la realización de diseños estructurados [Chu92,BoPi92].

Así, en la década de los sesenta aparecio el CDL y DDL, en los setenta el AHPL, ISPS, ISP', TIHDL y el Conlan y durante la década de los ochenta el ADLIB/SABLE, Zeus, VHDL y Verilog.

La primera aplicación de estos lenguajes ha sido la verificación de las arquitecturas de los diseños. Algunos HDLs no tienen la capacidad de modelar todos los diseños con un alto grado de exactitud. Otros como el HHDL, ISP' o VHDL sí implementan modelos temporales que pueden ser aplicados a cualquier tipo de estructura hardware.

Todos los HDL están soportados por un simulador. Esto permite que sistemas complejos y caros de implementar mediante prototipos puedan ser descritos mediante un HDL y verificados a través del proceso de simulación. El empleo de HDLs ha permitido que durante los últimos años haya cambiado el concepto que se tenía del proceso de diseño. Hoy en día, los diseñadores parten de la especificación de un diseño mediante captura de esquemáticos, realizando una descripción HDL o combinando ambas aproximaciones, para pasar luego al proceso de simulación y verificación.

Los dos últimos HDLs introducidos en el mundo del diseño han sido VHDL y Verilog. Por la importancia que estos dos lenguajes tienen actualmente vamos a comentar más en detalle sus prestaciones, sin entrar en su sintaxis, semántica, ni en detalles de implementación, por no ser relevantes en esta tesis.

VHDL:

VHDL (VHSIC Hardware Descripctión Language) comenzó a desarrollarse formalmente en 1983 bajo el auspicio del Departamento de Defensa de Estados Unidos. Originariamente estaba pensado para ser un mecanismo de comunicación de información de diseños digitales, pero ha llegado a ser un estándar internacional para la comunicación de diseños [Geus92]. A partir de 1985 fue creciendo el interés por establecer un estándar de lenguaje de descripción hardware IEEE. Este proceso culminó en 1987 cuando IEEE adoptó la versión 1076 de VHDL como estándar.

En VHDL cualquier circuito lógico por muy simple o complicado que sea, está representado como un **design entity**. Un design entity consiste realmente en dos tipos de descripciones diferentes: la descripción del interfase (**architectural design**) y uno o más cuerpos arquitecturales (**architectural bodies**).

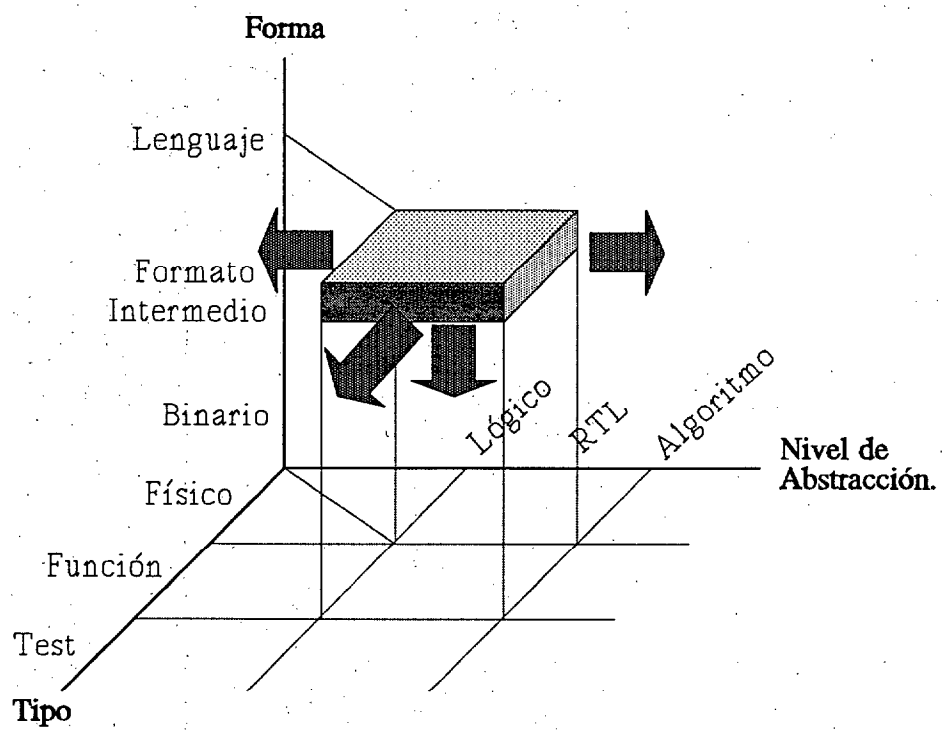

Fig. 2.9 Espacio tridimensional de información de un diseño.

Podemos clasificar la información asociada con el proceso de diseño de productos electrónicos de acuerdo a tres atributos: tipo, abstracción y forma. En la Fig. 2.9 podemos ver el espacio tridimensional formado por estos tres ejes. En el eje de tipo, podemos señalar como la información de diseño está clasificada como física, función y test. Cada uno de los tipos de información del diseño puede estar representado en varios niveles de abstracción como circuito, switch, lógica, registro de transferencia, algoritmo, *performance*, etc. El eje de forma denota cómo los ejes de tipo y abstracción indican la naturaleza de la información del diseño. Los mecanismos de representación pueden estar clasificados jerárquicamente como lenguajes de alto nivel, formatos intermedios o especificaciones binarias.

El área sombreada de la figura indica el espacio de información del diseño ocupada originariamente por VHDL. Las flechas indican las cuatro actividades de investigación en las

que se está intentando expandir el espacio ocupado por VHDL.

* Modelado de datos.
* Modelado analógico y a nivel de switch.
* Modelado en comportamiento y prestaciones
* Modelado para test.

VERILOG :

Verilog comenzó a comercializarse en 1985 aunque continuó desarrollandose durante algunos años más. Los lenguajes que más influyeron sobre el desarrollo de Verilog fueron HILO-2 y Occam [ThMo91, BoPi92].

Hoy en día incluye un completo conjunto de primitivas, que incluyen puertas lógicas, primitivas definibles por el usuario, switches NMOS y PMOS, nets triestados, y otros mecanismos que le convierten en un simulador switch y gate level de altas prestaciones y exactitud.

Las construcciones estructurales básicas permiten realizar asignaciones en las que expresiones que engloban valores de registros y nets, son asignadas a nets. También los resultados de cálculos hechos sobre registros y nets pueden acumularse en registros, siendo éstos los mecanismos básicos de descripción de comportamiento.

Al igual que VHDL el diseño consiste en un conjunto de módulos, cada uno de los cuales tiene un interfase de entradas/salidas con el resto de los módulos, y una descripción de su funcionalidad, que puede realizarse de forma estructural, de comportamiento o una mezcla de ambas.

Además de poder declarar procesos concurrentes que comienzan a ejecutarse automáticamente en el tiempo cero, podemos controlar el flujo temporal de los procesos utilizando retrasos, eventos y estados de espera.

2.3 Entornos de diseño CAD.

2.3.1 Introducción general.

El entorno CAD representa una colección de mecanismos y facilidades (librerías de programación, lenguajes, bases de datos, interfases de usuarios,...) en muchos niveles diferentes de abstracción. Por ejemplo, un entorno debe ser usado para configurar un conjunto de herramientas y desarrollar interfases apropiados para soportar captura de esquemáticos, simulación, verificación temporal y generación de test para gate arrays, edición de layout simbólico, compactación de layout, verificación para CMOS.

Un entorno bien diseñado proporciona muchos niveles de abstracción, y en general, todas estas capas están accesibles por los desarrolladores de herramientas y los integradores de sistemas CAD para su uso. Eso significa que un integrador de sistemas puede elegir usar facilidades de alto nivel proporcionados por el entorno, facilidades de bajo nivel, e incluso

llamadas al sistema proporcionadas por el sistema operativo en sí mismo, si fuera necesario.

Un entorno está construido sobre los servicios proporcionados por el sistema operativo el cual incluye facilidades para manejar y organizar ficheros, ejecutar programas, comunicarse con otros ordenadores a través de redes electrónicas y comunicarse con los usuarios del sistema.

En la última década el ordenador ha pasado a ser un miembro activo en el proceso de diseño. Por ejemplo de ser una herramienta de diseño gráfico interactivo de layout manual, ha pasado a ser usado para layout automático. De ser una herramienta de verificación de una función lógica a través de la simulación, a ser usado en la síntesis lógica automática de circuitos [KuhOh90]

Las áreas en donde el CAD ha tenido mayor impacto han sido tres: diseño físico, síntesis lógica y el trabajo en equipo. La evolución de las metodologías CAD se ha dado también en este mismo orden.

Diseño Físico:

En el área de diseño físico las herramientas de diseño automáticas e interactivas han llegado a ser indispensables para realizar el layout de los circuitos VLSI. El tiempo medio de los diseños se ha reducido de meses a semanas, a días o incluso a horas [HuKuh85][Oht86][MiSaA87]. Las técnicas de diseño *standard cell* y *gate array* prevalecen en diseño ASIC. Incluso para circuitos a medida de gran volumen como son los microprocesadores, las herramientas CAD para *placement*, *floorplanning* y *routing* son esenciales. A medida que los circuitos incrementan su complejidad, aumentan sus dimensiones y disminuye el tamaño de los dispositivos, es claramente necesario software de diseño y nuevos algoritmos más complejos. Por ejemplo, el estilo de diseño *sea-of-gates* [KuhOh90][SeSa86][SeLee87], necesita algoritmos eficientes que manejan cientos de miles de puertas. En este caso los algoritmos utilizados anteriormente basados en *simulated annealing* u otros métodos aleatorios llegan a ser completamente insuficientes.

Síntesis Lógica:

La tarea de síntesis parte de una especificación del comportamiento requerido de un sistema y de un conjunto de restricciones y objetivos que se han de satisfacer, y trata de encontrar una estructura que implemente el comportamiento, siempre y cuando se satisfagan todas y cada una de las restricciones y objetivos.

Por comportamiento entendemos la manera con que un sistema o sus componentes interactúan con su entorno. Por estructura nos referimos a un conjunto de componentes lógicos interconectados que formando el sistema; típicamente está descrita como netlist. Por último, la estructura debe ser mapeada en un diseño, esto es, una definición detallada de cómo el sistema está construido en una tecnología.

Trabajo en equipo:

Como la complejidad de los circuitos crece, los diseñadores no solo emplean un gran número de herramientas CAD diversas, sino que trabajan también en equipos para poder realizar sus tareas. Por este motivo, ha crecido el interés en entornos CAD que

permitan no solo la integración de múltiples herramientas CAD en un único sistema, sino que además proporcionen un medio de comunicación entre las diversas herramientas CAD y los diseñadores.

2.3.2 Revisión de los entornos CAD comerciales como soporte de arquitecturas paralelas.

A la hora de enfrentarse con la necesidad de realizar un diseño concreto, tenemos que acudir a los entornos CAD comerciales. En este apartado comentaremos las pautas principales de algunos de estos entornos, para luego comentar cómo se adaptan éstos a las especiales características de las arquitecturas objeto de esta tesis.

Como ejemplo de los entornos CAD comerciales de ayuda al diseño electrónico, comentaremos la líneas principales de trabajo de estos dos, así como cuales son las herramientas principales que disponen. En particular describiremos las líneas maestras de dos entornos ampliamente utilizados en el mundo del diseño electrónico como son el de Cadence y el de Mentor Graphics. Además de estos dos entornos comentaremos brevemente algunas característica interesante del entorno IDaSS.

2.3.2.1 Entorno de Diseño de Cadence.

OPUS™ IC Design System, es el nombre comercial que recibe el entorno Cadence. Es un entorno de trabajo orientado fundamentalmente al diseño de circuitos integrados, y está compuesto de herramientas de captura de esquemáticos, simulación, edición de layout y verificación. Todas ellas corriendo sobre el núcleo del sistema de diseño que se denomina *Design Framework™*.

El Design Framework:

Sobre la arquitectura del Design Framework corren las distintas herramientas que forman el conjunto del sistema. Los principales puntos sobre los que se apoya su filosofía de trabajo son los siguientes:

* **Entorno de diseño consistente.**

  Esto significa que todas las herramientas utilizan un el mismo interfase de usuario, y además existe una base de datos unificada en la que las distintas herramientas almacenan toda de información de un diseño.

* **Cooperación entre las distintas herramientas.**

  El entorno permite que existan múltiples herramientas corriendo concurrentemente con resultados intermedios para otras.

* **Entorno abierto.**

Es un sistema abierto de forma que el usuario está facultado para incorporar nuevas herramientas y adaptar otras existentes a sus propias necesidades. Los mecanismos que permiten al usuario acceder a los parámetros del entorno son dos:

- **SKILL** es un lenguaje de programación de alto nivel que permite el acceso a los recursos del Design Framework. Usando SKILL podemos adaptar la operatividad del sistema, permitir el acceso a la base de datos o adaptar las operación de una herramienta Cadence, etc.

- **OSS** (Open Simulation System) es un sistema que permite conectar simuladores al entorno de Cadence. Se pueden preparar ficheros de estímulos, ejecutar un simulador o visualizar los resultados de un simulador externo mediante formas de onda.

* **Portabilidad.**

El concepto de portabilidad nos indica la capacidad para que diversos usuarios que se encuentran sobre distintas máquinas puedan trabajar sobre partes de un mismo diseño compartiendo sus datos.

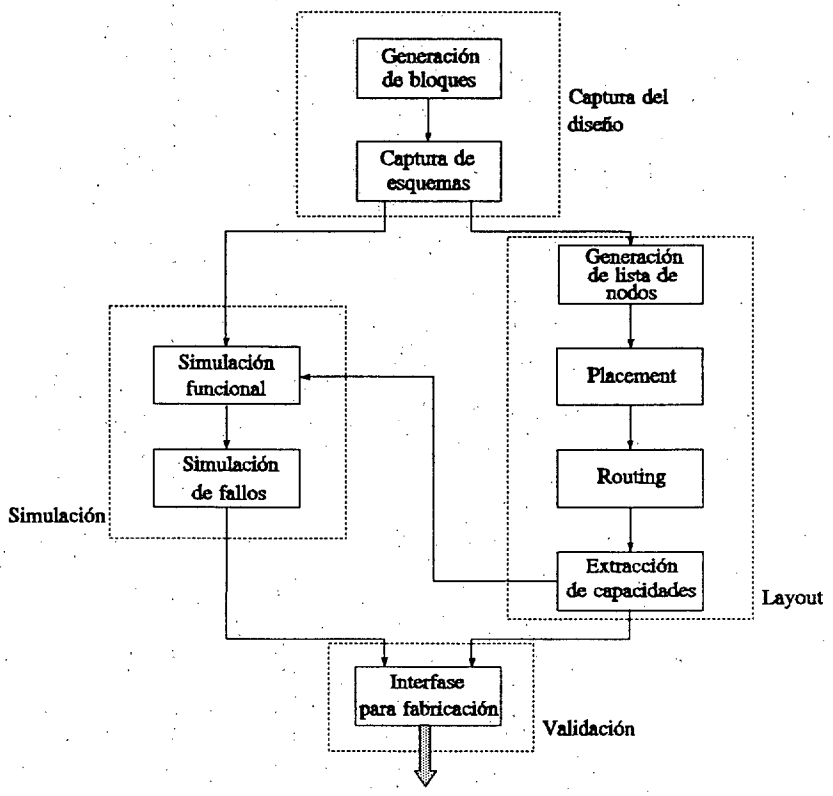

Fig. 2.10 Flujo general de diseño y simulación.

Herramientas de captura de esquemas y simulación:

Las herramientas de captura y simulación nos cubren las primeras etapas del diseño. La simulación y las tareas de edición de layout están estrechamente relacionadas tal como podemos apreciar en la Fig. 2.10.

* **Captura del diseño:** El editor de esquemáticos nos permite preparar la información necesaria para el resto de las herramientas. Además desde él se realiza el acceso a otras herramientas, como puede ser el caso de de las herramientas de síntesis lógica de Synopsys.

* **El entorno de simulación:** Se permite la simulación con diversos simuladores, pero siempre a través del mismo interfase para facilitar las tareas de aprendizaje. Desde el interfase, vamos a poder capturar el esquema, generar la netlist, describir los vectores de test, simular y mostrar graficamente los resultados.

* **Lenguajes de simulación y Test:** Una vez que la información del diseño ha sido introducida, se ha de preparar ésta de forma adecuada para los distintos simuladores que son soportados por Cadence. Estos simuladores son SILOS, Verilog, HiFault, System HILO, SPICE, etc..

  La creación de estímulos para los diversos simuladores también está unificada ya que se utiliza el lenguaje STL (Simulation and Test Language) como lenguaje común encargándose el compilador de generar los diversos formatos específicos para el simulador que se vaya a utilizar.

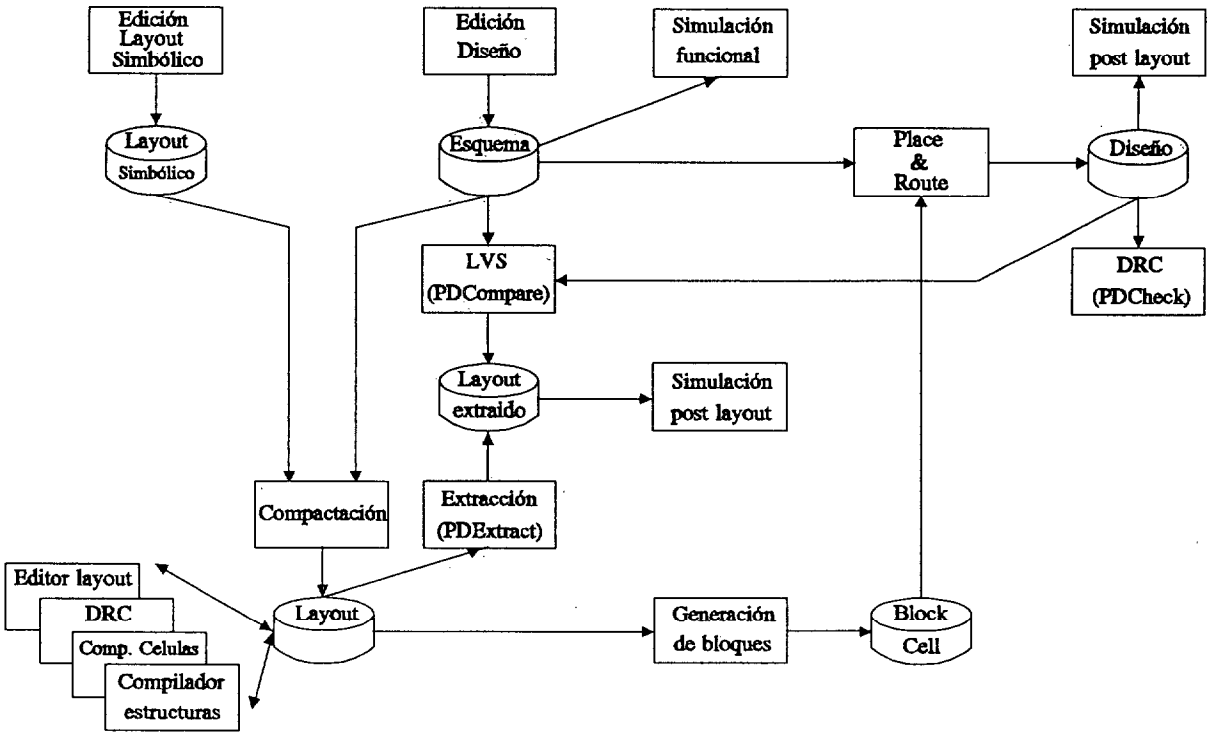

Fig. 2.11 Proceso de diseño de circuitos integrados con Cadence.

Herramientas de Layout:

El proceso de diseño físico de un circuito integrado sigue dentro de este entorno las pautas señaladas en la Fig. 2.11. Comenzando por la edición de esquemas o de layout simbólico, se pasa a utilizar las distintas herramientas de generación, edición y verificación de layout, para continuar con el proceso de verificación de los layout y pasar a la etapa de posicionamiento,

conexionado y verificación final. De una forma más detallada los distintos tipos de herramientas disponibles son los siguientes:

* **Editores de Layout:** Dentro del paquete de herramientas encontramos las distintas herramientas de diseño del layout de circuitos integrados. Estos editores están ayudados por una serie de herramientas adicionales como los editores simbólicos, los compactadores de layout, sintetizadores de layout, verificadores de reglas eléctricas o reglas de diseño, etc.

* **Traductores de datos:** Existen un conjunto de distintos programas traductores que permiten convertir información procedente de otros entornos de trabajo. Estos traductores son fundamentalmente de dos tipos: traductores de EDIF y traductores de layout.

* **Herramientas de Place & Routed:** Después del proceso de captura de esquemáticos y de edición de layout, tenemos que enfrentarnos con la tarea de posicionamiento de las células del diseño y de su posterior conexionado.

* **Gate Array Ensemble:** Es la herramienta encargada tanto del diseño de gate-arrays convencionales como de mar de puertas. Las tecnologías disponibles son CMOS, Bipolar, ECL, BiCMOS y GaAs, permitiendo hasta un máximo de 6 capas de interconexiones.

* **Cell Ensemble:** Es un sistema de propósito general para el diseño de circuitos integrados basados en células. Realiza por lo tanto el placement y routing de diseños de células estándar.

* **Block Ensemble:** Esta herramienta permite realizar el placement de bloques de dimensiones no uniformes.

* **Tancell:** Es una herramienta de placement y routing automática para diseños basados en células estándar.

* **Floor Planner:** Es una herramienta de layout que facilita el diseño de circuitos al poder realizar el floor-planing de bloque incluso antes de que estos estén diseñados.

* **ModuleMaker:** Esta herramienta que facilita el desarrollo de generadores de módulos parametrizables. El producto se distribuye en dos paquetes, uno denominado ModuleMaker Development package, que permite el diseño y construcción de generadores de módulos y el Modulemaker Executable package que permite ejecutar los generadores creados.

Herramientas de Verificación:

Durante el proceso de diseño de circuitos integrados debemos verificar los diseños realizados para detectar lo antes posible los errores que se puedan haber cometido. Existen dos grandes grupos de herramientas de verificación: las herramientas interactivas y las standalone.

* **Verificación Interactiva:** Nos dan información interactivamente, de forma que pueden correr simultáneamente con las herramientas de diseño, verificadores son:

- PDcheck:
- PDextract:
- PRE (Parasitic Resistance Extraction):
- ERC (Electrical Rules Checker):
- PDcompare:

* **Verificación standalone:** Es el caso del paquete de verificación de Dracula III, que ha de correr externamente. Está compuesto por la siguientes herramientas:

  - Dracula III.
  - Dracula DRC/ERC.
  - Dracula LVS/LVL.
  - Dracula LPE/LPEPRO.
  - Dracula/PRE.
  - Dracula_Access.
  - Dracula LayDe.
  - Dracula DRE.
  - Dracula PLOT.
  - Dracula PG/EII.

2.3.2.2 Entorno de Diseño Mentor Graphics.

Mentor Graphics ofrece una familia de soluciones EDA (Electronic Design Automation) compuesta de aplicaciones software, estaciones de trabajo y periféricos que permiten incrementar la productividad de los procesos envueltos en las distintas fases del diseño electrónico [Men92]. El software está desarrollado de forma totalmente modular que permite adaptarlo a las necesidades de los usuarios. Las distintas herramientas están agrupadas en paquetes software denominados estaciones (*stations*). Cada estación está orientada hacia un aspecto específico del proceso de diseño. Por ejemplo, el paquete de herramientas para el diseño de circuitos impresos se denomina "Board Station" y está compuesto de las aplicaciones necesarias para la captura de esquemáticos, simulación y para posicionamiento y conexionado de circuitos (*placement and routing*).

Todas las aplicaciones usan el mismo entorno denominado **Falcon Framework**. El entorno de trabajo Falcon establece un entorno común para las operaciones de las distintas aplicaciones y suministra un método común de manejar diseño y compartir información de la base de datos con otras aplicaciones de Mentor Graphics o productos de terceras partes.

En la versión 8.1 las estaciones que se distribuidas son:

* **Falcon Framework Station.** Es el mínimo núcleo necesario para poder correr cualquier otro producto Mentor Graphics, para manejar datos y diseño, para ver documentación en línea (online), y para desarrollar y ejecutar programas de soporte de decisiones (decision support programs).

* **Board Station.** Permite el diseño de circuito impresos. Está compuesto por captura de esquemáticos, diversos interfases y las aplicaciones de posicionado y conexión de PCB.

* **Hybrid Station.** Permite la creación de una gran variedad de circuitos híbridos.

* **IC Layout Station.** Es un entorno integrado para la realización del layout de circuito integrados complejos.

* **IC Layout EX Station.** Tiene todas las características de la estación de Layout, pero además incorpora aplicaciones para chequeo de reglas de diseño, comparadores entre esquemático y layout, y un interfase al programa DRACULA de chequeo de reglas de diseño.

* **Idea Station.** Está compuesta por Design Architect con los editores de esquemáticos, símbolos y VHDL, más el simulador lógico QuickSimII.

* **MCM Station.** Permite la creación, placement y routing, análisis térmico y aplicaciones para la generación de ficheros de salida para el desarrollo de módulos multichip.

El ciclo de diseño.

Tradicionalmente el método empleado en el desarrollo de productos consiste usualmente en realizar una serie de pasos secuenciales e independientes, fundamentalmente fraccionando el desarrollo del producto en aplicaciones. Con esta aproximación secuencial, muchas de las decisiones que debemos tomar deben de esperar hasta que se complete el ciclo de diseño.

Mentor Graphics aborda el problema del tradicional ciclo de diseño a través de una arquitectura que incorpora los principios de *ingeniería concurrente* (concurrent engineering) y *diseño concurrente* (concurrent design). Esta arquitectura se llama Mentor Graphics Concurrent Design Environment (CDE).

* El principio de *ingeniería concurrente* permite al usuario tomar decisiones importante tan pronto como sea posible en el ciclo de diseño. Esta nueva aproximación permite que todas las fases de un diseño estén integradas. Con estos procesos se permite un fácil intercambio de información entre varias tareas a través del ciclo de diseño.

* El principio de *diseño concurrente* permite que varias tareas estén superpuestas, de forma que los pasos posteriores puedan comenzar antes de haber concluido los previos. Usando diseño concurrente, mejoramos el trabajo en paralelo, realizamos el análisis del desarrollo de productos antes, y tenemos una mayor coordinación en los grandes proyectos.

Características del Falcon Framework.

Tal como puede apreciarse en la Fig. 2.12 el entorno CDE, constituye un entorno común a todas las aplicaciones y presenta como aspectos destacados los siguientes elementos:

* **Common User Interface (CUI).** Es un interfase común a todas las aplicaciones. Está desarrollado sobre OSF/Motif, aunque también corre sobre OpenWindows.

* **Design Manager.** Es una herramienta que simplifica el manejo y coordinación de datos. Permite la visualización, operación y organización de las aplicaciones y los

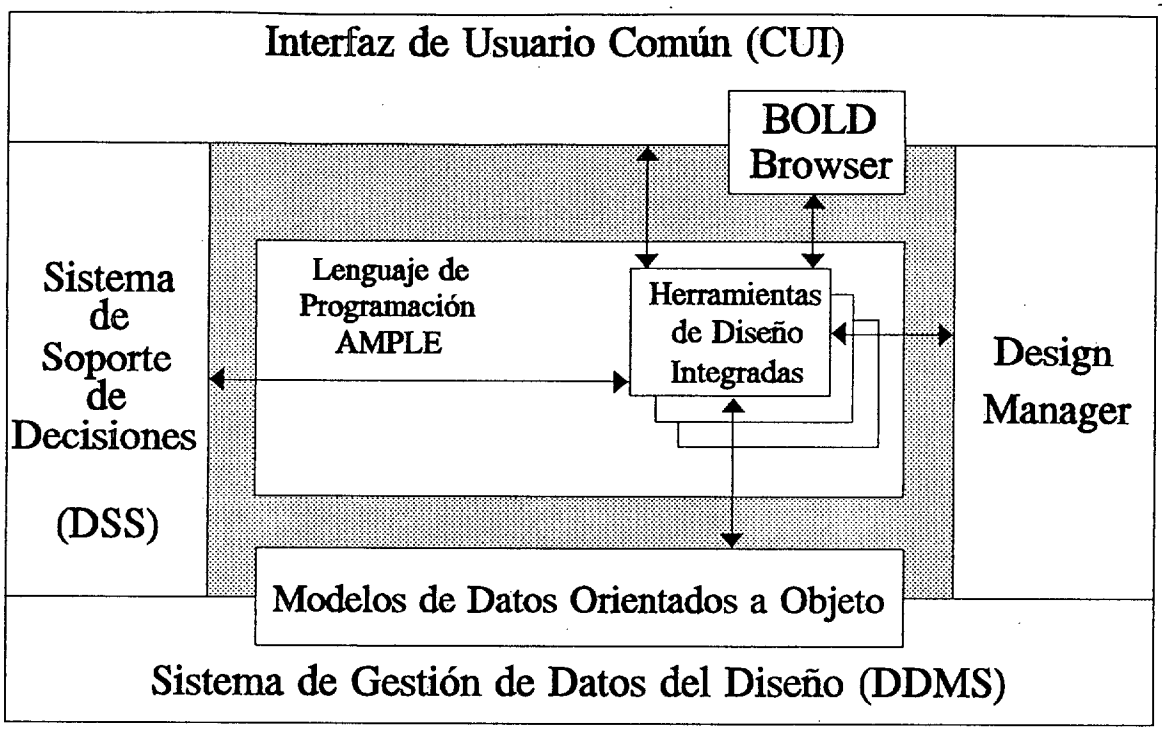

Fig. 2.12 Falcon Framework.

datos. El Design Manager presenta los diseños, objetos del sistema y aplicaciones como iconos.

* **Decision Support System (DSS).** Es un entorno que soporta el proceso de tomas de decisiones y compartición de información a través del proceso completo de desarrollo de productos. Contiene una hoja de cálculo, paneles de control gráficos y diversos interfases que permiten el acceso a los datos del diseño.

* **AMPLE.** Es un lenguaje de programación procedural que permite la adaptación y comunicación a lo largo de todas las aplicaciones.

* **Common Data-Modeling.** Es un modelo de datos común que permite el uso de modelos de datos orientados a objeto único en todas las aplicaciones.

* **Design Data Management System (DDMS).** Realiza la gestión a bajo nivel de los datos objetos, tales como gestión de múltiples versiones y atributos de cada uno de los objetos.

* **Multi-Ventana.** Esta característica permite tener diversas aplicaciones simultáneamente sobre la misma estación de trabajo.

* **BOLD.** Es una herramienta de información y ayuda del sistema que permite acceder a todos los documentos de Mentor Graphics desde un CD ROM.

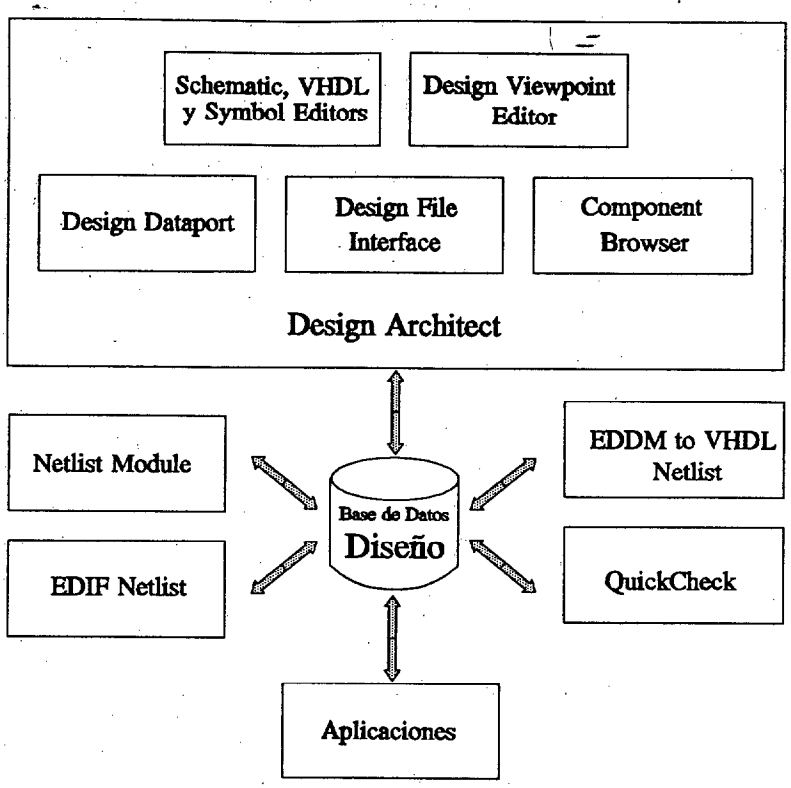

Fig. 2.13 Aplicaciones de captura de esquemáticos.

Proceso de Captura de un Diseño.

El proceso de captura de un diseño suministra todas las capacidades necesarias para capturar el diseño en los niveles arquitecturales abstractos y en los niveles de lógica detallada y circuitos. En la Fig. 2.13 se muestra varios de los procedimientos

* **Design Architect** es el entorno de diseño que permite la creación de símbolos, esquemáticos, modelos VHDL. Combinando el editor de esquemáticos con el editor VHDL se proporciona templates de lenguaje para la rápida y exacta entrada de construcciones VHDL.

* **Design Viewpoint Editor (DVE)** es una aplicación interactiva que permite al usuario asignar y cambiar las reglas que configuran sus diseño. Además permite verificar el diseño, ver modelos, crear, editar y manejar retro-anotaciones.

* **Component Interface Browser (CIB).** Un Component interface es un método de atar informaciones de modelos a un componente. El Component Interface Browser (CIB) proporciona una manera rápida y conveniente de inspeccionar y cambiar los contenidos de un interfase de componente. CIB también ayuda en la gestión de componentes de una librería.

* **QuickCheck.** Es una herramienta de verificación adaptable por el usuario. Se compone del Name Checker y del Electric Rule Checker que se encargan comprobar las instancias, pines y propiedades de los nodos, así verificar las reglas de conexionado eléctrico.

* **Netlisters.** Una netlist es una lista de todas las informaciones de conectividad de un

diseño. Design Architect crea una base de datos, la cual únicamente identifica cada nodo, instancia y propiedades de un diseño. Los Netlister son programas que analizan y traducen la información producida por Design Architect a o desde varios formatos de netlist.

El VHDL Netlister traduce los datos de un diseño de Mentor en el formato VHDL de forma que éstos puedan ser utilizados como datos de entrada a otras aplicaciones de simulación que no sean de Mentor Graphics. Existe otro Netlister como es el EDIF Netlist Read o el EDIF Netlis Write que permiten leer o escribir en formato EDIT respectivamente.

* **Procedural Interfaces.** Los Interfases Procedurales son conjuntos de procedimientos o funciones de Pascal o C que permiten a los programadores experimentados acceder y manipular la base de datos de un diseño.

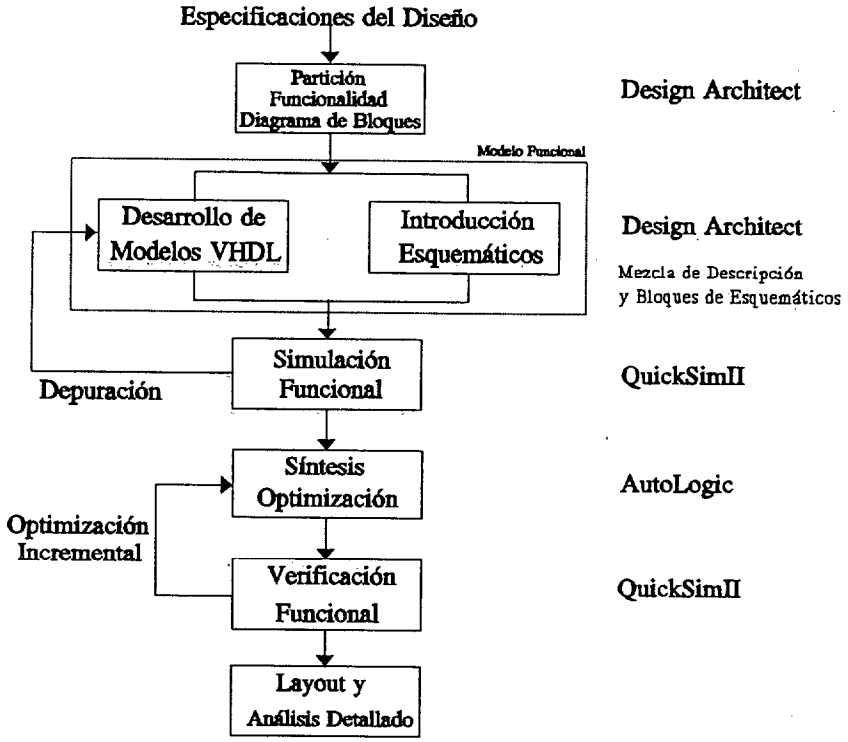

Fig. 2.14 Proceso de diseño Top-Down con síntesis lógica.

El proceso de síntesis lógica.

Mentor Graphics ofrece productos de síntesis para dispositivos lógicos programables (PLD), circuitos integrados (ICs) y circuitos integrados de aplicación específica (ASIC). Estos productos son básicamente el **PLDSyntesis**, que en realidad una opción al proceso de captura que permite generar de forma eficiente una o más PLDs que implementan la parte del diseño que se indique, y la familia de productos de Autologic que incluye:

AutoLogic VHDL. Accepta como entrada una descripción de la norma IEEE 1076 standard y la sintetiza al nivel de registros de transferencia. Es independiente de la tecnología, lo cual permite capturar, simular y sintetizar el diseño completo antes de realizar la implementación física. Cuando el diseño está finalizado se puede realizar

la lógica con cualquier combinación de implementación de ASIC, PLD o FPGA empleando AutoLogic para la optimización e implementación dependiente de la tecnología.

AutoLogic. AutoLogic es una herramienta de optimización del diseño que produce una netlist específica de una tecnología a partir de una netlist genérica.

Proceso de Simulación.

En la Fig. 2.14 se muestra el ciclo completo de diseño *top-down* que incluye una etapa intermedia de síntesis lógica. La herramienta principal de simulación es el simulador lógico QuickSimII, que es capaz de manejar simultáneamente muy diversos modelos de componentes (Tabla II.3). El paquete completo de herramientas de simulación es el que se presenta a continuación:

* **SimView** es el interfase de usuario común a las aplicaciones de simulación de Mentor Graphics y es usado para configurar y analizar una simulación. Cuando se invoca a un simulador, SimView puede ser utilizado para desarrollar un conjunto de estímulos para su uso posterior en otra sesión de simulación, o para analizar formas de onda que han sido producidas por la salida de una aplicación de simulación.

* **Charting** es una función de post-análisis incluida en SimView y es usada para presentar gráficamente los datos de simulación en un formato de alta resolución versátil. Se permite cambiar los ejes de un diagrama para resaltar una curva de datos particular, o para poner varias curvas sobre el mismo diagramas. Charting permite múltiples ventanas de diagramas para permitir una fácil organización de diferentes tipos de datos de acuerdo a las necesidades.

* **QuickSimII** es un simulador lógico interactivo que permite verificar la funcionalidad de diseño producidos con Design Architect. QuicSimII es un simulador tipo timing-wheel de 12-estados, que puede simular varias tecnologías como CMOS, TTl, ECL y otras.

* **Circuit PathFinder (CPF).** Es una aplicación de análisis temporal que realiza análisis temporal a nivel de transistores en circuitos integrados digitales CMOS y NMOS. Operando a una velocidad interactiva sobre una base de datos de un circuito de cualquier tamaño, ofrece las siguientes posibilidades de extracción de camino críticos, estimación temporal de caminos, generación de netlist de caminos lista para simular e integración de parásitos extraídos de layout en análisis temporales.

* **QuickGradeII.** es un evaluador estadístico rápido de fallos que ayuda en el desarrollo de conjuntos efectivos de vectores de entrada que testean el circuito para la detección de fallos tipo *stuck-up*.

* **QuickPath.** Es una herramienta de análisis estático que realiza varios tipos de análisis temporal en un esquemático de diseño. Cuando se invoca a QuickPath se pueden ver las hojas de esquemático en una ventana y seleccionar todo o porción de un esquemático para el análisis. Después de generar los resultados del análisis, se pueden crear ventanas adicionales que presentan esos resultados de forma gráfica o textual.

* **System-1076**. Este es el compilador y simulador de VHDL basado en la IEEE std 1076-1987. System-1076 está integrado en el entorno de Mentor Graphics. Los modelos VHDL pueden ser desarrollados usando el compilador de System-1076 y entonces ejecutados utilizando un simulador digital, tal como QuickSimII.

Tabla II.3 Técnicas empleadas por Mentor Graphics para el modelado de componentes.

Modelos de esquemáticos
Son los esquemáticos creados utilizando Design Architect, en los que se han distribuido las característica temporales en la hoja sobre los modelos que nos gustaría simular. Estos modelos son fáciles de realizar, pero ocupan mucho espacio de disco y la velocidad de simulación no es tan buena como en las otras alternativas.

Modelos QuickPart Schematic
Son modelos digitales creados procesando una hoja de esquemático con el compilador **quickpart** antes de la simulación. El resultado de la simulación es una primitiva digital que puede ser instanciada en un esquemático o en otro modelo QuickPart. El modelo compilado ocupa menos espacio de disco y simula más rápido que los modelos basados en hojas de esquemáticos.

Modelos QuickPart Table
Creados procesando un fichero de tabla QuickPart. Estos ficheros son ficheros ASCII similares a tablas lógicas. Describen los diferentes estados de un dispositivo basándose en sus pines de entrada y salida. Se compilan utilizando el compilador **qpt**, ocupando menos espacio y simulando más rápidamente que los modelos basados en hojas de esquemáticos.

Modelos Lógicos (LM).
Algunos dispositivos LSI o VLSI, tales como microprocesadores o son tan difíciles o consumen tanto tiempo para modelarlos con software que es más efectivo modelarlos utilizando un modelador hardware que permitan integrar estos componentes en la simulación. Durante la simulación los estímulos y las salidas son intercambiadas entre la estación de trabajo y la máquina de modelado hardware. La familia de LM es un sistema de modelado hardware que trabaja a través de red de terceras partes, pero que está comercializada y soportada por Mentor Graphics.

Modelo de Tabla de Memoria.
Son creados procesando un fichero de interfaz ASCII que describe el comportamiento de la lógica de control de acceso a memoria y acciones de un dispositivo de memoria en array. Estos modelos ofrecen una forma rápida y exacta de modelar los dispositivos ROM y RAM estática.

Modelos Incorporados (Builtin Model).
Son los modelos lógicos internos de todas las puertas lógicas básicas, incorporados en el simulador QuickSimII.

Behavioral Language Model (BLM).
Son modelos que utilizan ficheros fuentes de procedimiento o funciones de Pascal o C para describir un dispositivo. Estos ficheros son compilados con el compilador apropiado. Bien escritos, estos modelos simulan más rápidamente que otros y son de gran exactitud.

SmartModels.
Los modelos SmartModels son modelos BLM suministrados por terceras partes. Los SmartModels proporcionan al diseñador modelos de microprocesadores, memorias y otros componentes de uso general para simulación a nivel de sistema.

Modelos VHDL.
Las descripciones VHDL creadas usando Design Architect pueden ser compiladas utilizando System-1076 que proporciona un modelo que puede ser utilizado por el simulador QuickSimII.

2.3.2.3 Entorno de Diseño de IDaSS.

IDaSS (**I**nteractive **D**esign **a**nd **S**imulation **S**ystem) es una herramienta para el diseño a nivel arquitectural de sistemas digitales [Ver90, Ver92]. Se fundamenta en la realización simultánea de los procesos de simulación y captura de esquemáticos, de esta forma el diseñador puede implementar y comparar diferentes arquitecturas de sistemas complejos.

El diseño con IDaSS es fácil ya que diseño y simulación están integrados. Cada modificación es simulada instantáneamente, de forma que cada acción del diseñador puede ser verificada inmediatamente. Las librerías están compuestas con macro-células predefinidas, tales como microcontroladores, o otros bloque como RAM, ROM y operadores complejos.

Está integrado con ASA Silicon Compiler de forma que a partir de una descripción IDaSS se pueda generar directamente un ASIC. ASA automáticamente realiza la síntesis lógica, análisis temporal y generación automática del layout para suministrar un camino directo desde el diseño al silicio.

2.3.2.4 Sistemas de control de procesos.

En otros campos que no son exactamente el del diseño electrónico existen herramientas que nos ayudan a realizar tareas de control y monitorización de procesos. Por ejemplo, en el campo industrial es conocidos por todos los múltiples paquetes disponibles en el mercado de software tipos SCADA. Estos paquetes nos permiten monitorizar de forma gráfica sobre la pantalla de un ordenador los esquemas unifilares de una instalación industrial, diagramas de cuadros eléctricos de maniobras, equipos de medidas de diversos parámetros físicos (temperatura, humedad, caudal, capacidad de un depósito, etc.), estados de trabajo de elementos (válvulas, accionadores, motores eléctricos, turbinas, etc.), alarmas y eventos, o diversas gráficas sobre la evolución de los parámetros de mayor interés.

Incluso algunos de los paquete actuales permiten utilizar dispositivos multimedia mediante los cuales podemos ver, sobre una de las ventanas de nuestro entorno, una imagen de video que no muestra un diagrama de una instalación industrial, sino la propia instalación industrial sobre la que estamos actuando. Adicionalmente algunas de las alarmas o eventos que se producen, además de indicarlos gráficamente, se pueden generar de forma auditiva a través del dispositivo de audio incorporado al ordenador.

2.4 Conclusiones sobre lo expuesto.

Después de realizar una exploración sobre los tres campos abordados en esta tesis (arquitecturas paralelas, lenguajes de especificación hardware y entornos CAD de diseño) se extraen las siguientes conclusiones:

Los entornos CAD generales como son CADENCE o Mentor Graphics, no hacen ningún tratamiento especial para el caso de diseño de arquitecturas con alto grado de paralelismo. Su principal objetivo es suministrar un buen conjunto de herramientas más o menos integradas entre sí que permiten al diseñador abordar cualquier tipo de diseño. El entorno de usuario en estos caso es normalmente amigable, aunque para realizar el diseño de arquitecturas paralelas tengamos que acudir a la herramienta de captura de esquemáticos tradicional para realizar la descripción estructural de nuestro diseño, con la consiguiente perdida de tiempo que esa actividad conlleva.

Existen otros entornos especializado en los que el trabajo del diseñador es más cómodo y sencillo, ya que están pensados para trabajar con un determinado tipo de tecnología o estrategia de diseño. En estos entornos, como son el caso de algunos compiladores de silicio, el trabajo está muy enfocado al tipo de circuitos que vamos a obtener al final del proceso.

Salvo escasas excepciones en la inmensa mayoría de sistemas CAD, los resultados de simulación podemos verlos en la formas tradicionales de diagramas de estados. Existen en cambio otras herramientas que a la vez que realizamos el proceso de captura del diseño, vamos situando junto a los símbolos eléctricos símbolos como voltímetros u otros elementos de medida que durante el proceso posterior de simulación nos irán indicando gráficamente los valores de las señales a las que están conectados (este es el caso de algunos simuladores analógicos-digitales como el simulador MIXsim de Sierra Semiconductor).

De todo lo expuesto, se extrae la conclusión de que es de gran utilidad una herramienta que nos permita de una forma rápida y sencilla describir arquitecturas paralelas para una rápida evaluación de éstas. A continuación una vez vista la descripción y que es correcta, se generarán de forma automática los ficheros y procesos necesarios para incorporar la arquitectura descrita al entorno de diseño seleccionado, en el cual se procedería a concluir el diseño físico del circuito o circuitos con las herramientas estándar del entorno CAD al alcance del diseñador. Además es de gran interés que durante el proceso de simulación, podamos visualizar de forma alternativa los resultados de simulación en un entorno gráfico equivalente al que podemos encontrar en los sistemas de monitorización y control de procesos industriales.

Capítulo 3.

Descripción del sistema propuesto: EASAP.

3.1 Introducción.

En este capítulo damos una primera descripción del entorno creado con pequeñas muestras de su uso. En el capítulo 4 se da su aplicación al diseño de varios ejemplos completos, lo que se ha hecho también como evaluación. Y en el capítulo 5 se describe en profundidad la implementación del entorno, dando todos los conceptos y detalles de interés.

3.2 Planteamientos iniciales.

La realización de esta tesis se planteó para intentar dar respuestas a una serie de preguntas que nos hacíamos a la hora de abordar el diseño de arquitecturas digitales VLSI. Como un gran porcentaje de las arquitecturas que se necesitan poseen características de regularidad y paralelismo, nos preguntamos lo siguiente:

¿ qué tipo de herramientas hacen falta para ser capaces de concebir, diseñar y simular estos tipos de arquitecturas. ?

¿ qué tipo de herramientas podemos encontrar en el mercado. ?

¿ qué es lo que me gustaría utilizar ?

Ante este tipo de preguntas, las respuestas son claras. Existen innumerables entornos que nos permiten diseñar y simular cualquier tipo de arquitectura, lo único que tenemos que saber es qué queremos diseñar. Si conocemos lo que queremos diseñar, el resto es inmediato si se cuenta con los medios apropiados. Es decir, si sabemos lo que queremos lo más importante ya lo tenemos, lo de menos es la herramienta que utilicemos para su realización. Ante esto, se evidenció la falta de un lenguaje de descripción de arquitecturas que recogiera este tipo de requerimientos, por lo que tuvimos que tener en cuenta los siguientes puntos:

a) Evidentemente podemos utilizar los lenguajes de descripción hardware como el VHDL como un entorno de simulación comercial. Esta solución es válida, pero si queremos adaptar el VHDL a las necesidades específicas de las arquitecturas objeto de estudio, nos hace falta un superset de éste para realizar la descripción. En este caso el software utilizado ya no sería estándar.

b) Otra solución es un conversor de nuestro superset VHDL a un set estándar.

c) Otra solución es crear un lenguaje nuevo con su consiguiente conexión a un

entorno de simulación comercial o a un entorno propio de simulación como es el caso de los lenguajes específicos que se han desarrollado.

En nuestro caso, se optó por la solución c), pero por supuesto la idea no era desarrollar un nuevo simulador, sino conectar con uno ya existente y visualizar los resultados de éste de una forma más eficiente para el diseñador. Ya que existen distintos entornos de simulación la idea es desarrollar un sistema rápido y flexible de descripción. Para la creación de un entorno abierto y flexible, se ha optado por emplear el lenguaje C para la descripción de las arquitecturas, lo que nos permite crear una base de datos de la que incluir y extraer información.

3.3 Aplicación al campo de la formación de diseñadores.

Uno de los objetivos fundamentales que también se pretende con esta tesis es el desarrollar un método de formación de diseñadores. Las aportaciones que podemos realizar en este campo podemos resumirlas en dos puntos:

1) Enseñanza de algoritmos y arquitecturas existentes.

* Se requiere por parte del alumno las siguientes características:

  Conocimientos técnicos básicos sobre técnicas de simulación digital, diseño de circuitos VLSI y manejo de herramientas CAD con las cuales realizar los diseños. No basta tener unos buenos conocimientos sobre los temas comentados anteriormente, sino que además hay que tener una buena capacidad de comprensión espacial, ya que hay que imaginarse el flujo de información de los distintos datos a través de los arrays.

* Por parte del profesor:

  Hay que ser capaz de explicar los procesos de movimiento de información de las distintas arquitecturas. Esto obliga a recurrir a elementos como el uso de transparencias para poder explicar cómo se realiza la transmisión de la información entre los distintos elementos. La animación de las secuencias se hace del todo imprescindible en este tipo de explicaciones.

2) Diseño de circuitos.

Una vez que ya conocemos el problema que queremos resolver con una arquitectura con alto grado de paralelismo (como pueden ser las arquitecturas sistólica), nos enfrentamos al problema de cómo se realiza el circuito que implementa la arquitectura seleccionada.

Aquí nos enfrentamos a la toma de una decisión sobre el tipo de tecnología que tendremos que utilizar. Típicamente emplearemos una solución en VLSI, ya que se ajusta muy bien a los objetivos de la realización de estructuras con un alto grado de repetición.

A la hora de la realización física de circuitos VLSI descritos con nuestras herramientas, hay que tener en cuenta la problemática particular de la tecnología, sobre todo con los problemas de consumo, de distribución de las líneas de reloj a lo largo del circuito, de ubicación de componentes, etc.

3.4 Tipos de arquitecturas objeto de estudio.

Las arquitecturas digitales sobre cuales vamos a centrar nuestra atención, podemos dividirlas en tres grandes grupos con características bien diferenciadas desde nuestro punto de vista: Arquitecturas Regulares Sistólicas, Arquitecturas Regulares No Sistólicas y Arquitecturas No Regulares.

Aplicación a Arquitecturas Regulares Sistólicas.

Las particulares características de un sistema de este tipo dificultan la exacta comprensión de los fenómenos que se ocasionan. Nos vamos a centrar en los siguientes puntos o casos que podemos encontrar.

1) Dado un algoritmo (clasificable dentro de los que se pueden mapear), necesitamos diseñar un sistema sistólico que lo resuelva. En este caso hay que recurrir a los distintos tipos de algoritmos clásicos de partición y mapping de un algoritmo en una estructura concreta.

2) Dada una estructura (alguna de la conocidas) ver como puede ser modificada para ajustarse a un problema particular.

3) En el caso de tener un problema que no posee soluciones semejantes ni disponemos de un algoritmo que nos pueda encontrar la solución de una forma automática, debemos acudir a la imaginación para solventar nuestro problema particular. En este punto hay que considerar que la mayoría de los métodos que podemos emplear no conducen a una solución única, sino que nos dan un abanico de posibles soluciones.

Es de resaltar que nuestro objetivo final es la implementación de una estructura VLSI que nos solucione el problema. Por este motivo a la hora de la selección de una de las distintas alternativas hay que considerar los problemas asociados al diseño final full-custom o semi-custom. La selección final sólo será posible si contamos con algún indicador que nos de una idea de área, consumo o velocidad del circuito final.

Como conclusión encontramos que no es fácil alcanzar una solución satisfactoria mediante un procedimiento automático, por lo que la experiencia del diseñador es uno de los factores primordiales a tener en cuenta. Como no existe un método sistemático de alcanzar siempre una solución, es necesario disponer de un sistema de ayuda que nos facilite las tareas tediosas de evaluación de las distintas arquitecturas.

Aplicaciones a Arquitecturas Regulares No Sistólicas.

Dentro de este apartado encontramos todas aquellas arquitecturas cuyo grado de regularidad estructural es alto, pero que no podemos englobarlas dentro de la

clasificación de sistólicas. Como ejemplos de estas arquitecturas podemos encontrar sumadores, multiplicadores, memorias, etc.

Aplicaciones a Arquitecturas No regulares.

Dentro de esta clasificación englobamos el resto de arquitecturas que no se encuentran en los dos puntos anteriores, o bien son arquitecturas en las que algunos de sus elementos son regulares pero, el conjunto no se puede considerar regular. Como ejemplos tenemos microprocesadores, microcontroladores, y circuitos ASIC en general.

En este tipo de arquitecturas el lenguaje LDAP no aporta una gran ventaja frente al método clásico de descripción estructural que son los esquemáticos. En este caso el punto de entrada a nuestro entorno no será **GeneSis**, sino el conversor apropiado al entorno CAD en el cual estemos trabajando.

3.5 Conceptos generales.

Vamos a revisar en este punto todo lo relativo a los conceptos con los que se trabaja en el entorno **EASAP**. Será necesario tener claros estos conceptos para poder seguir con seguridad las explicaciones que se den en los siguientes apartados acerca de los comandos del lenguaje LDAP, verdadero "front-end" del entorno. Pero antes de entrar en los conceptos principales del entorno dedicaremos el siguiente apartado a exponer el origen, motivación y objetivos de nuestro lenguaje de descripción.

3.5.1 El lenguaje LDAP.

Se trata de un lenguaje desarrollado de forma que permita la descripción de los componentes de un sistema microelectrónico o, incluso, de cualquier sistema genérico cuyos componentes puedan ajustarse a los utilizados por este lenguaje.

Nuestro lenguaje presenta como una de sus características principales, que lo diferencia de otros tipos de lenguajes, la capacidad de dar información sobre cuál es la situación física de las líneas de entrada y salida de cada una de las células.

Esta característica en la descripción de una célula nos permite que al definir un array de células, el interprete de este lenguaje se hace cargo automáticamente de realizar las conexiones entre células adyacentes.

El nombre de este lenguaje, LDAP, proviene de las siglas de ***Lenguaje de Descripción de Arquitecturas Paralelas***. Inicialmente este lenguaje se pensó únicamente para su uso en la forma de macrofunciones del lenguaje C. Sin embargo, en la actualidad, el lenguaje LDAP se presenta en sus dos versiones desarrolladas para compilación e interpretación.

Si bien, como se ha comentado, originalmente sólo se podía hacer uso de este lenguaje mediante la confección directa de un fichero fuente C, con el formato adecuado para su compilación posterior con las librerías aportadas por esta herramienta, la necesidad de desarrollar la versión interpretada del LDAP surgió para dar respuesta a aquellos diseñadores que desearan generar las descripciones y simulaciones de sus sistemas sin necesidad de tener que aprender la sintaxis de un lenguaje de programación (a pesar de que el LDAP ha sido

implementado de tal forma que, como se verá más adelante, prácticamente no se necesita conocer en profundidad la sintaxis de este lenguaje para poder conseguir ya buenos resultados).

Los comandos básicos del lenguaje, tienen como principal objetivo la descripción de la arquitectura que pretendemos simular. Los elementos básicos que se manejan en estos tipos de arquitecturas, son las células y los arrays de células. Por lo tanto en nuestros diseños encontraremos fundamentalmente células, arrays, conexiones entre células de arrays y conexiones entre el mundo exterior y las células de nuestro diseño. Adicionalmente tenemos capacidad de definir una serie de elementos gráficos que nos sirven para visualizar el proceso de simulación.

Todos los elementos que forman una descripción LDAP, aunque estén definidos unos dentro de otros de forma jerárquica, al posicionarlos sobre el área de trabajo se tratarán, a efectos de selección, como una descripción plana. A efectos de transformaciones gráficas de desplazamiento, rotación o escalado se mantendrá la descripción jerárquica (Fig. 3.1).

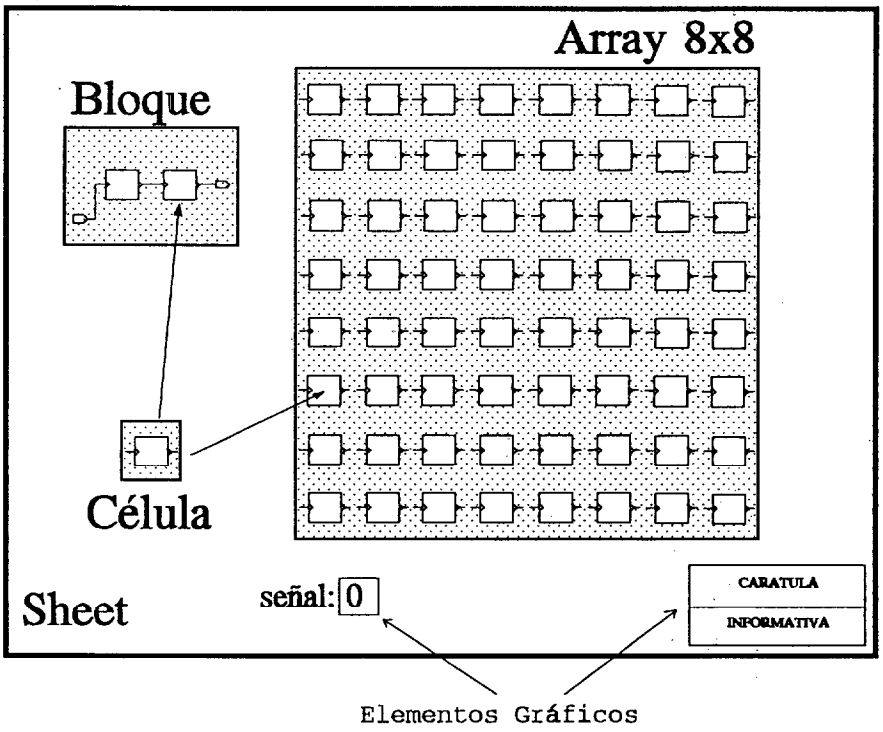

Fig. 3.1 Area de trabajo.

Con este objeto, por tanto, se desarrolló la versión interpretada del LDAP que proporciona al usuario un interfase de alto nivel desde el cual poder realizar las descripciones de sus sistemas sin la necesidad de tener que descender a conocer la sintaxis de un lenguaje de programación en particular. Como se comentará en el apartado dedicado al analizador ***lexer***, se llega incluso a dar al diseñador la opción de, a partir de un fichero con la descripción de su sistema con el LDAP interpretado, generar un fichero de salida con el mismo contenido de información pero escrito en LDAP compilado, es decir, un fichero fuente C ya preparado para su compilación directa. Si este es el caso, el propio intérprete se encargará de hacer automáticamente todas las transformaciones necesarias en el formato de los comandos y usos de ficheros externos (con información de simulación, como se verá en su momento) para

adaptarlos al formato exigido por el LDAP compilado. Este punto se desarrollará en el apartado dedicado al intérprete ***lexer*** (analizador léxico generado con **lex**).

3.5.2 Elementos de un sistema.

En nuestro recorrido por los conceptos a definir en nuestra herramienta, el primer paso debe ser la definición de aquellos tipos de elementos con los que el diseñador contará para realizar la descripción de los sistemas que desee generar. Se definen cinco tipos básicos de elementos:

* células
* arrays
* bloques
* elementos gráficos de dibujo
* sheets

Conceptualmente, **célula** es el elemento básico de posicionamiento en un sistema y a partir del cual se define el contenido de los restantes elementos del sistema. Internamente, puede estar formada por cualquier número de puertas lógicas y señales; sin embargo, en la mayoría de las ocasiones, su conectividad interna será transparente a la herramienta y deberá ser indicada externamente para su simulación (a través de un fichero de datos externo indicado por el usuario en su descripción).

Los elementos **array** y **bloque** son muy similares, diferenciándose básicamente en que aquél sólo contendrá células, mientras que un bloque podrá estar constituido tanto por células como por arrays e, incluso, por otros bloques. Hay que indicar aquí, que el concepto de array no se limita al concepto de '*conjunto de elementos de igual tipo y características*', sino que se amplía a la posibilidad de que los elementos que lo constituyen sean de clases diferentes. Ello permite al usuario definir un único array que recoja células de diferentes características externas, en lugar de verse obligado a realizar la definición de un tipo de array para cada conjunto de ellas. Tenemos aquí un caso muy típico de un array de células en un sistema microelectrónico, en el cual las células de la periferia tienen características ligeramente diferentes que las diferencian de las del interior. El LDAP permite dar la definición de un único array que las englobe a todas.

Los **elementos gráficos de dibujo** nos permitirán definir líneas, cajas, circulos, textos, y estructuras para visualizar los valores de las variables de las células. Pueden estar declaradas dentro de cualquier de los elementos descritos anteriormente.

Finalmente, el **sheet** es quien contiene a todos los demás, es decir, podrá estar formado por cualquier combinación de los otros tres elementos. El usuario puede especificar la descripción de tantos sheets como desee en el mismo sistema. En principio, cada uno será tratado como un elemento aislado de los demás sheets (si los hubiera). Sin embargo, existe la posibilidad de asociar un sheet a la representación interna de una célula de ese mismo sistema, de forma que, si bien se mantienen informaciones de conectividad internas independientes, pueda establecerse esta relación ante un simulador externo y así conseguir una simulación completa de todos los elementos del sistema que se relacionan entre sí.

3.5.3 Declaración de elementos.

Los elementos que se han descrito en el apartado anterior son los elementos básicos que reconoce el LDAP. Sin embargo, son tan sólo **tipos básicos** a partir de los cuales el diseñador define sus propios tipos de elementos combinando los elementos básicos de cualquiera de las formas permitidas. Estos tipos, ahora definidos por el diseñador, serán los que éste podrá utilizar al definir un elemento real sobre un sistema (en realidad, sobre un sheet del sistema).

Las normas del lenguaje LDAP obligan a que cualquier elemento que el diseñador utilice para definir uno de los elementos *compuestos* (array o bloque) o para describir un sheet del sistema, debe ser de un tipo de elemento que él haya definido antes de esa operación de definición o descripción. Ahora bien, no hay ningún problema por realizar las definiciones de elementos básicos en cualquier orden, siempre que los elementos que van a ser contenidos en otros elementos sean definidos antes que aquellos elementos que los contienen.

Los elementos que pueden ser contenidos y los que pueden contener a otros son los que se muestran en la Tabla III.1 donde, además, también se muestran los elementos que puede contener cada uno de los elementos básicos reconocidos por el LDAP. En la Fig. 3.2 tenemos un ejemplo de sintaxis LDAP en donde se encuentran definidos distintos tipos de elementos.

***Tabla III.1**: Relación entre elementos básicos*

ELEMENTO	CONTENIDO POSIBLE
Célula	Sheet*, Elementos Gráficos
Array	Células, Elementos Gráficos
Bloque	Células, Arrays, Bloques, Elementos Gráficos
Sheet	Células, Arrays, Bloques, Sheets*, Elementos Gráficos

(*) *Ver modelos de visualización.*

Obsérvese la referencia que se hace, en el caso de que el elemento contenido sea un sheet, a los modelos de visualización. Se han marcado estos dos casos como especiales dado lo atípico de la situación que representan. Una célula puede contener un sheet, si bien esto será de forma *indirecta* y a través de un camino alternativo a la definición normal de un elemento contenido en otro. Todo ello se explicará en detalle en el apartado **Modelos de Visualización**.

3.5.4 Modelos de visualización.

Para cada uno de los elementos definidos por el usuario como parte de la definición de un sistema, se puede realizar la definición de un modelo de visualización de ese elemento. Este modelo consistirá en un fichero de algún tipo que contenga una visión del elemento afectado. No tiene por qué ser un modelo de visualización gráfica, aunque éste sea el caso más típico, sino que puede ser información de cualquier tipo que se tenga acerca del elemento en cuestión.

```
USE default_level;
SIMUL V_VHDL = 'output.vhdl';
...
CELL cell_1 {
        INOUT { ... }
        INTERNAL { ... }
        FUNCTION { ... }
        MODEL { ... }
}
...
ARRAY array_1 { ... }
...
ARRAY array_n { ... }
...
BLOCK block_1 {
        PLACE { ... }
        CONNECT { ... }
}
...
BLOCK block_n {
        DRAWS { ... }
}
SHEET sheet_1 {
        DRAWS { ... }
        PLACE { ... }
        CONNECT { ... }
}
...
AUTO { cell_1; .... }
```

Fig. 3.2 Estructura de una descripción LDAP.

El LDAP reconoce actualmente los modelos que se aprecian en la Tabla III.2, Tabla V.3, varios de ellos basados en simuladores actualmente en el mercado. De estos modelos, podemos extraer algunos cuyo comportamiento es especial y nos permitirán cierto tipo de operaciones.

Tabla III.2: Modelos reconocidos.

NOMBRE	MODELO DE...
V_CHDL	Descripción funcional de la célula en C
V_WAVES	Descripción de los vectores de simulacion
V_LAYOUT	Layout de una célula en CIF
V_VERILOG	Modelo VERILOG de simulacion
V_HILO	Modelo SYSTEM HILO de simulacion
V_VHDL	Modelo VHDL de simulacion
V_SIMON	Modelo para lenguaje LIV de SiMon
V_SHEET	Modelo para SHEET
V_PINEXT	Modelo para pin externo
V_PININT	Modelo para pin interno

En primer lugar, es importante el modelo V_SHEET. Cuando en la Tabla III.1 se hacía referencia a sheets contenidos en elementos del tipo célula o sheet, era a través de la definición de este modelo con lo que se conseguía tal asociación. Para aclarar ideas veamos el mecanismo por pasos. En principio, si el usuario intenta situar dentro de un sheet un elemento cuyo tipo corresponde a otro sheet del sistema, se topará de frente con un error

denunciado por el LDAP. De igual forma, tampoco tiene la posibilidad de definir directamente un sheet dentro de una célula. Sin embargo, mediante el empleo del modelo V_SHEET se solucionan ambos problemas al mismo tiempo.

Si al definir una célula se le asocia un modelo V_SHEET, se le está indicando al LDAP que esa célula dispone de una descripción interna que se corresponde plenamente con la descripción que se dé en el sheet del sistema cuyo nombre se da a continuación de V_SHEET (ver sintaxis del comando para modelos de visualización). Ello, a todos los efectos, equivale a situar un sheet dentro de una célula. Pero, no solamente eso, ya que si esa célula se posiciona dentro de otro sheet, como suele ser lo habitual, se estará indicando también el posicionamiento en éste de aquel sheet que actúa de modelo para la célula.

De cara a la información a enviar a los simuladores externos, si estamos en la situación en que hemos situado un sheet dentro de otro (a través de un modelo de célula, por supuesto), habremos de indicarle a aquellos tal relación entre ambos sheets. Para ello, el diseñador que defina un modelo V_SHEET para una célula, deberá proceder de forma que los pines de E/S de esa célula coincidan con los pines externos del sheet o, de otro modo, no se podrá establecer relación alguna entre los sheets.

Para señalar los pines externos de un sheet se hace uso de otro de los modelos definidos en la Tabla III.2, Tabla V.3. Se trata, en esta ocasión, del modelo V_PINEXT. Este modelo se asocia a aquella célula que se desea que actúe como pin externo de E/S del sheet en que se posiciona. Para mantener la relación entre los pines externos del sheet y los definidos en la célula de quien es modelo, el diseñador deberá posicionar estas células de modelo V_PINEXT con un nombre coincidente con el del pin al que quieren representar. Por ejemplo, sea una célula con un modelo llamado *Test* del tipo V_SHEET (por lo tanto, el diseñador deberá describir aparte un sheet con nombre *Test*), y de la que se definen los pines *A*, *B* y *C*. En el sheet *Test* deberán posicionarse, al menos, tres células cuyos nombres lógicos sean *A*, *B* y *C*, y que sean de un tipo de célula al que se les asocia un modelo V_PINEXT. En la Fig. 3.3 se muestra cómo quedaría esto en un fuente del LDAP interpretado.

```
/* Define tipos para las células que serán pines de E/S */
CELL input {
        INOUT { pin : IW }
        MODEL { V_PINEXT = 'Entrada' }
}
CELL output {
        INOUT { pin : IW }
        MODEL { V_PINEXT = 'Salida' }
}

/* Define tipo de célula jerárquica */
CELL Sistema {
        INOUT { A:IW; B:IW; C:OE }
        MODEL { V_SHEET = 'Test' }
}
....
....
/* Describe sheet asociado a la célula jerárquica */
SHEET Test {
        ....
        PLACE {
                A = input:[0,0]:[10,10]:0;    /* Posiciona los pines de E/S */
                B = input:[5,10]:[10,10]:0;
                C = output:[20,0]:[10,10]:0;
        }
        ....
}
```

Fig. 3.3 Ejemplo de uso de los modelos V_SHEET y V_PINEXT.

El otro modelo asociado a éste será V_PININT. Este se incluye simplemente para recoger el caso de varios modelos V_SHEET definidos para una misma célula (los llamaremos *sheets hermanos*) y entre los cuales hay una serie de pines que los comunican entre sí pero que no se relacionan con los de la célula de la cual son modelos estos sheets. Por ello se les denomina *pines internos*.

3.5.5 Autoconexionado de pines y generación automática de símbolos.

Un concepto de gran utilidad para el diseñador es el de la ***autoconexión***. Con esta se libera a aquél de una gran cantidad de trabajo mecánico para realizar la conexión de un número de señales que podría ser bastante elevado. Así, durante la definición de una célula, el diseñador puede realizar un emparejamiento entre algunos de los pines de la misma mediante el empleo del comando ***Def_Con()*** o el operador '>' (para el compilador o el intérprete, respectivamente).

Así, cuando se crea un array en el que se sitúan células de igual tipo, una junto a otra, el objetivo final es realizar la conexión entre los pines de cada célula y los de cada una de sus vecinas. Para ello, al haber relacionado los pines de la célula entre sí, el software dispone de la información necesaria para llevar a cabo de forma automática esta serie de conexiones para cada célula. Sin embargo, debe tenerse en cuenta que un array podrá estar formado por células de diferente tipo. Ello no supone ningún problema, ya que la autoconexión de un pin se realizará incluso con el de una célula vecina de diferente tipo, siempre y cuando el nombre de ese pin sea igual que el de aquél que debería autoconectarse con éste si sus células fueran de igual tipo.

Por tanto, el diseñador no tendrá necesidad de especificar la conexión de los pines *internos* de un array, ya que estos serán conectados automáticamente al cerrar la descripción del sheet. No se olvide que sólo serán autoconectados aquellos pines que así hayan sido señalados por el diseñador.

Sin embargo, no es ésta la única aplicación que se saca de la operación de emparejamiento entre dos pines. Así, cuando se genera la salida para un visualizador gráfico, hay que darle a éste las coordenadas donde se desea ver cada elemento. Para ello, lo lógico es realizar el cálculo de éstas de acuerdo a cómo se asocia cada pin con los demás. Mediante la información dada por el emparejamiento, el diseñador puede indicar que pines deben presentarse gráficamente enfrentados. Ello, al visualizar el sistema, ayudará a la visión rápida de las conexiones que hay entre células de igual tipo.

A la hora de generar de forma automática el símbolo de una célula, el sistema tiene en cuenta tres cosas:

* tipo de señal (entrada, salida, bidireccional o global).
* punto cardinal por el que entra o sale de la célula (N, S, E, W, NE, NW, SE o SW).
* señales que se encuentran autoconectadas.

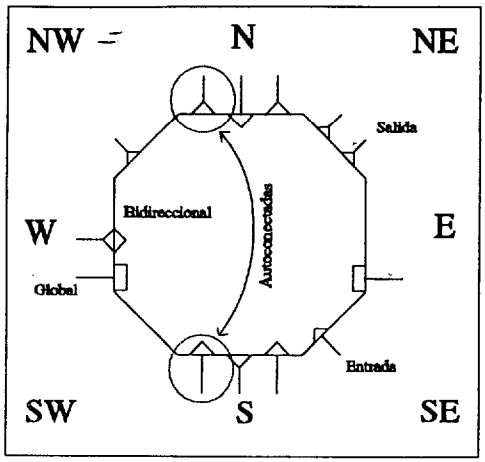

Fig. 3.4 Ejemplo de creación de un símbolos.

3.6 Descripción de las herramientas y lenguajes utilizados.

En este apartado vamos a realizar una descripción del entorno de herramientas que se plantea en esta tesis. En vista del tipo de arquitecturas sobre las cuales vamos a trabajar, necesitamos un lenguaje de descripción hardware que nos ayude en las tareas de concepción y simulación de estas arquitecturas, además necesitamos poder visualizar y controlar la simulación de la arquitectura de una forma eficiente. Para lograr estos objetivos se van a exponer los siguientes puntos:

a) Proceso de Trabajo.
b) Lenguaje LDAP Interpretado.
c) Lenguaje LDAP Compilado.
d) Lenguaje de Descripción Visual (LDV).
e) Lenguaje de Intercambio de Variables (LIV).
f) Sistema de Generación (GeneSis).
g) Sistema de Monitorización (SiMon).

a) Proceso de Trabajo.

El proceso de trabajo con nuestro entorno comienza con la descripción de la arquitectura que queramos simular. Para ello utilizando el lenguaje LDAP podemos describir a nivel estructural la arquitectura objeto de estudio. Con el lenguaje LDAP lo que estamos describiendo de una forma óptima, es las interconexiones entre los distintos elementos de la arquitectura propuesta. No estamos describiendo la funcionalidad de estos elementos, sino la forma en que están conectados. La funcionalidad de los elementos la describiremos en el lenguaje que utilicemos para simulación.

Con el empleo del LDAP, no sólo podremos realizar una descripción estructural, sino que además podremos declarar una serie de elementos gráficos que nos ayuden durante el proceso de visualización y control de la simulación.

El LDAP puede ser utilizado en su versión interpretada o compilada.

En la versión interpretada partimos de un fichero ASCII que contiene la descripción de la arquitectura paralela objeto de estudio.

En la versión compilada partimos de la descripción de nuestro problema utilizando un conjunto de funciones C de librería que nos permiten la especificación de la arquitectura. El fichero de la descripción hay que compilarlo con las librerías del entorno para así obtener un código ejecutable válido.

Una vez que el código está compilado, se ejecuta generando directamente la arquitectura descrita.

La descripción visual de la arquitectura se realiza mediante el Lenguaje de Descripción Visual (LDV) y el intercambio de variables desde y hacia el simulador se realiza mediante la utilización del Lenguaje de Intercambio de Variables (LIV).

El proceso general de trabajo puede ser gestionado desde la herramienta GeneSis, que se encarga de generar los distintos formatos de ficheros para cada una de la herramientas propuestas. La relación entre las distintas herramientas mencionadas queda claramente descrita en la Fig. 3.5.

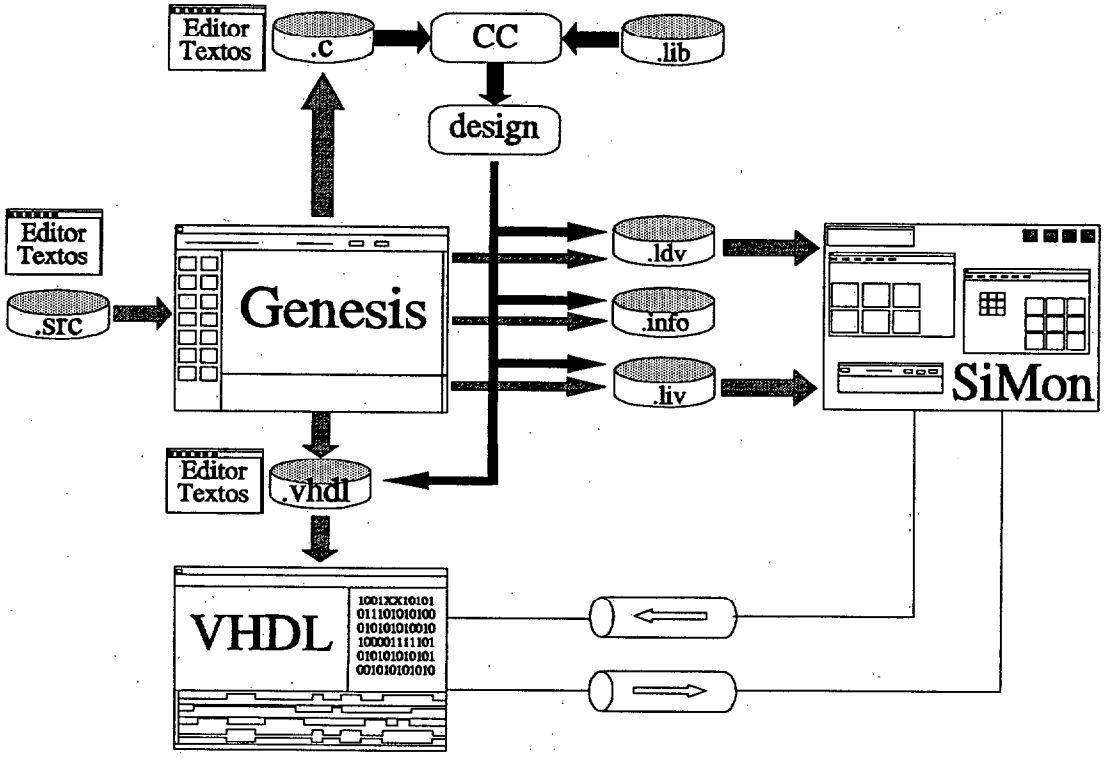

Fig. 3.5 Vista general del entorno de trabajo.

b) Lenguaje LDAP Interpretado.

Mediante la utilización de un lenguaje de descripción de arquitecturas podemos de una forma rápida describir un sistema que pretendemos simular. Los comandos básicos del lenguaje, tienen como principal objetivo la descripción de la arquitectura que pretendemos simular. Los elementos básicos que se manejan en estos tipos de arquitecturas, son las células y los arrays de células. Por lo tanto en nuestros diseños encontraremos fundamentalmente células, arrays, conexiones entre células de arrays y conexiones entre el mundo exterior y las células de nuestro diseño.

Nuestro lenguaje presenta como una de sus características principales, que lo diferencia de otros tipos de lenguajes, la capacidad de dar información sobre cuál es la situación física de las líneas de entrada y salida de cada una de las células. Esta característica en la descripción de una célula permite que al definir un array de células, el compilador de este lenguaje se haga cargo automáticamente de realizar las conexiones entre células adyacentes. Como la mayoría de las arquitecturas que estamos tratando están compuestas de arrays regulares de células iguales, el lenguaje permite con una sola línea de código realizar cientos de conexiones automáticamente, liberando al diseñador de las tareas más mecánicas de especificación de conexiones. Adicionalmente tenemos capacidad de definir una serie de elementos gráficos que sirven para visualizar el proceso de simulación.

Las conexiones entre células de distintos arrays, han de ser especificadas por el diseñador. Este trabajo ineludible, es sin embargo aliviado por una sintaxis muy potente, que mediante la utilización de comodines (wild cards), permite realizar numerosas conexiones con solo el empleo de algunas instrucciones muy escuetas.

El intérprete de este lenguaje está desarrollado sobre la herramienta **lex** de Unix, y se encarga del análisis del fichero de entrada para luego proceder a llamar a las funciones del LDAP compilado que se explicará en el siguiente punto.

c) Lenguaje LDAP Compilado.

La versión para compilación del lenguaje LDAP habrá de seguir la sintaxis del lenguaje C y permitir el empleo de éste de forma interactiva con aquél. Los ficheros fuentes del LDAP para compilación serán así, igualmente, ficheros fuente para el compilador del lenguaje C.

La implementación en esta forma del LDAP compilado permitirá no sólo el empleo, lógicamente, de todas las primitivas de este lenguaje desarrollado, sino que también se podrá hacer un uso intensivo de todas las funciones que nos proporciona el lenguaje C en sus librerías. De esta forma, un fichero fuente para el LDAP compilado podrá tanto limitarse al empleo exclusivo de sus propias primitivas como extenderse al empleo de todas las facilidades de programación disponibles en C.

Las primitivas del lenguaje LDAP serán, en definitiva, macrofunciones desarrolladas en lenguaje C y que permitirán la descripción de cada uno de los elementos del sistema. La sintaxis en el empleo de estas primitivas se desarrollará de tal forma que se permita su manejo sin más que disponer de conocimientos mínimos acerca de la sintaxis propia de las funciones del C. Con esta implementación se podrán obtener ya resultados satisfactorios sin más que hacer uso de las primitivas del LDAP compilado.

Las primitivas que constituyen el LDAP compilado pueden considerarse divididas en tres grupos:

De generación de estructuras de datos

Son todas aquellas que utilizará el diseñador para describir las características de los elementos del sistema, así como de los componentes internos de los mismos. Su ejecución conduce a la generación de las estructuras de datos en memoria que mantendrán todas estas informaciones.

De generación de datos para simulación

Serán todas aquellas que se usarán por parte del diseñador para indicar aquellas informaciones encaminadas a la simulación del sistema en su conjunto y de cada uno de sus elementos. Su ejecución, normalmente, provocará la generación de ficheros de salida con los datos destinados a los simuladores externos.

De visualización gráfica del sistema

Serán todas aquellas primitivas, y datos aportados por el diseñador, que permitan la representación gráfica de los elementos que constituyen el sistema. Su ejecución provocará, en algunos casos, la generación de un fichero de salida en un lenguaje de descripción visual apropiado y, en otros, el almacenamiento temporal en memoria de ciertos datos relativos a las características visuales del sistema.

d) El Lenguaje de Descripción Visual (LDV).

El LDV es un lenguaje descriptivo jerarquizado el cual permite definir mediante elementos gráficos simples sistemas más complejos. Existen instrucciones que generan salida gráfica, salida textual e instrucciones que generan variables. Además, existen otras instrucciones las cuales son utilizadas para la definición de módulos elementales o primitivas que serán usadas en otras primitivas más complejas. Las arquitecturas descritas con este lenguaje son representadas gráficamente por SiMon.

e) El Lenguaje de Intercambio de Variables (LIV).

Mediante el uso de este lenguaje podemos comunicarnos con el programa SiMon para indicarle cuales son los cambios que han ocurrido en los valores de las variables declaradas en los distintos elementos de las arquitecturas descritas. Este lenguaje permite una comunicación bidireccional, es decir los cambios en las variables que sean introducidos por el usuario a través del entorno gráfico son comunicados también al proceso o procesos que se comuniquen con SiMon.

f) Sistema de Generación (GeneSis).

GeneSis permite la visualización descriptiva de los diferentes elementos que constituyen el sistema, así como la posibilidad de conseguir información adicional acerca de cada uno estos, en lo relativo a sus características internas y a su relación de conectividad con los demás elementos del mismo. Todas las utilidades que han sido desarrolladas en esta herramienta están a disposición del usuario de una forma eficiente y rápida, liberándolo de la necesidad de realizar una serie de tareas, repetitivas y mecánicas, destinadas a la obtención de las diferentes salidas que proporciona nuestro entorno.

La metodología de GeneSis se caracteriza no por realizar la simulación de la arquitectura, sino por permitir que ésta sea generada y simulada más eficientemente con simuladores estándar. GeneSis nos dará dos tipos de salidas, el fichero LDV para el programa de monitorización SiMon y un fichero con la descripción de la arquitectura que se haya realizado (VHDL por ejemplo).

El interfase proporciona, a su vez, los medios necesarios que permitan al diseñador el desarrollo de los ficheros fuentes de entrada con las descripciones del sus sistemas. Para ello habrá de disponerse, además de elementos del tipo editor de texto, de ventanas apropiadas para la ejecución de comandos directos del **UNIX**, mediante los cuales poder realizar toda la gestión de archivos que sea necesaria.

g) Sistema de Monitorización (SiMon).

SiMon es una herramienta de monitorización y control de procesos de aplicación general. Su función dentro de nuestro entorno es la de dar soporte al mecanismo de visualización y control del proceso de simulación de las arquitecturas descritas. Nos permite realizar las siguientes acciones:

Mediante el Lenguaje de Descripción Visual (LDV) podemos describir el diseño, arquitectura o sistema que pretendemos monitorizar y controlar.
Mediante el Lenguaje de Intercambio de Variable (LIV) podemos decirle a SiMon cuales son los cambios en las variables de nuestro diseño o preguntarle sobre el estado de cualquiera de ellas. Como cualquier sistema de control, notificará cualquier cambio o acción que el operario realice sobre las variables del diseño.

La comunicación entre SiMon y el proceso sobre el cual se realiza la acción de monitorización y control es bidireccional. El proceso notificará a SiMon los nuevos cambios de valores de variables y SiMon notificará al proceso las acciones realizadas por el operario durante la sesión de trabajo.

3.7 Métodos de trabajo con EASAP.

Hasta ahora, en los capítulos previos, se ha descrito la forma en que se han implementado las funciones que el usuario tiene a su disposición para la definición del sistema. Ahora nos vamos a centrar en cómo es la secuencia de trabajo que debe seguir el usuario para generar las estructuras adecuadas en la memoria.

Recuérdese que se ofrece la posibilidad de trabajar en dos niveles diferentes. Por un lado, se puede describir el sistema mediante el empleo de macrofunciones C presentes en la librería ***basico.c***. A este nivel, lo que el usuario hace, efectivamente, es crear un programa fuente C que habrá de ser compilado para generar el programa ejecutable correspondiente. La segunda opción de trabajo que se ofrece al usuario es la del lenguaje interpretado. Este está formado por un conjunto de palabras reservadas (o comandos) que actuarán de interfase en alto nivel para la llamada a las funciones C desarrolladas.

Según puede deducirse de lo dicho, tanto de una forma como de otra, se terminará haciendo uso de las funciones C de la librería ***basico.c*** para la definición de los elementos y la descripción de los sheets del sistema. La diferencia estriba en que, mediante el lenguaje compilado, es el propio usuario quien escribe directamente la llamada a estas funciones en el fichero fuente C a compilar, mientras que, mediante el lenguaje interpretado, lo que el usuario hace es crear un fichero ASCII con el lenguaje propio del entorno **GeneSis** y que será analizado gramaticalmente procesando la llamada a las funciones adecuadas de la librería ***basico.c***, pero sin que tenga la necesidad de conocer la sintaxis del lenguaje C.

3.7.1 Trabajo con LDAP compilado.

Como se ha comentado ya en repetidas ocasiones, en este nivel de trabajo se pretende que el usuario confeccione un fichero fuente para compilador C. Sin embargo, debe seguir unas normas estrictas para que el fuente resultante sea adecuado para su compilación y posterior ejecución.

A partir del empleo de las funciones proporcionadas, el usuario debe seguir unas normas a la hora de confeccionar el fichero fuente C para su compilación. En lo tocante al formato del fichero fuente, hemos de empezar por aquellos ficheros que deben incluirse al comienzo del mismo. Para ello se usará la sentencia ***#include*** del pre-procesador del compilador C, con la que se indica al compilador qué ficheros deben incluirse. El primero de estos ficheros contiene definiciones propias del lenguaje C y que serán necesarias si el usuario desea hacer uso de alguna de ellas (más información sobre éstas se puede encontrar en el Manual del Compilador C). El segundo de los ficheros incluidos consistirá en las definiciones propias de valores y variables definidas especialmente para el entorno **GeneSis**, y que serán necesarias para la descripción que se haga de cualquier sistema que desee generarse con esta herramienta. La inclusión de estas definiciones en el fichero ***basico.h*** libera al usuario de secuencias puramente mecánicas de líneas de texto a escribir.

Una vez que se han incluido estas líneas, hay que dar comienzo a la sección donde el usuario realiza la descripción del sistema. Para ello, deberá crear un función especial, la cual será reconocida por el entorno **GeneSis**, y que se denomine ***Main_Sheet()***. Esta función contendrá la descripción propiamente dicha del sistema, es decir, contendrá todos los comandos (funciones C) de la librería ***basico.c*** que sean empleados por el usuario para ello. El nombre de esta función, ***Main_Sheet()***, es muy importante y debe ser incluido invariablemente en el fichero fuente creado por el usuario, ya que es el nombre de la función que será llamada desde la función principal ***main()*** del programa ejecutable que se genere de la compilación y que será aportada por la librería con la que se compila el fichero fuente del usuario. Por tanto, si éste no usa este nombre para designar a la función desde la que se llamará a las funciones de definición de los elementos y descripción general del sistema, el proceso de *lincado* se abortará con un error debido a la falta de una función con este nombre.

En otras palabras, el usuario debe declarar una función de nombre ***Main_Sheet()*** y cuyo contenido será la descripción que se hace del sistema mediante las llamadas a las funciones de librería antes comentadas. Una vez dentro de esta función, y previo a la llamada a cualquier función de la librería, debe llamarse a ***Init_system()*** (también parte de ***basico.c***). Esta función (ver Cap. 5) inicializa diversas variables internas del sistema, reserva memoria para los diversos buffers transitorios que vayan a utilizarse y crea la estructura SYSTEM, que como se vio en su momento, es la raíz para todas las estructuras de datos que describen el sistema.

No declarar la función ***Main_Sheet()*** o no usar en primer lugar la función ***Init_System()*** conduce, en el primer caso, a un error en tiempo de compilación (el compilador, durante la fase de *lincado* no encuentra esta función y genera un error) o, en el segundo caso, en tiempo de ejecución (no hay memoria asignada para SYSTEM).

Una vez que se han cumplido estos requisitos, el usuario ya puede hacer uso de las funciones que desee para describir el sistema (siguiendo siempre la norma adecuada de empleo de cada función). Así, tras haber generado el fichero fuente, se procede a su compilación para ser

ejecutado. La compilación se realiza añadiéndole las librerías que contienen las funciones necesarias:

mainbas.c Contiene la función ***main()*** que llama a ***Main_Sheet()***, que habrá sido incluida por el usuario en el fichero fuente.

basico.c Contiene las funciones para generación de las estructuras de datos en memoria.

disco.c Contiene las funciones que permitirán el empleo de las operaciones con disco y generación de netlist para diferentes simuladores.

genldv.c Incluye todas aquellas funciones para la generación de ficheros con formatos LDV (*Lenguaje de Descripción Visual* para **SiMon**).

Teniendo en cuenta todo esto, la línea de orden al compilador, para que éste genere el fichero ejecutable deseado, sería bastante larga y tediosa de compilar. Para liberar al usuario de esta tarea, se ha preparado un fichero de procedimientos, ***gen***, que contiene esta línea y acepta como parámetro el nombre del fichero fuente del usuario (sin extensión). A continuación se muestra esta línea, donde la expresión '**$1**' se usa para indicar el parámetro que el usuario pasa al llamar a este fichero de procedimientos.

$ cc $1.c mainbas.o basico.o disco.o genldv.o -o $1

Con este fichero de procedimientos, la compilación del fichero fuente generado por el usuario se reduce a teclear el comando:

gs user_file

donde hemos supuesto que el nombre del fichero fuente es *'user_file'*.

3.7.2 Trabajo con LDAP interpretado.

Con esta opción de trabajo, liberamos al usuario de la necesidad de conocer la sintaxis del lenguaje C para poder describir el sistema. En esta ocasión, seguirá el formato dado por el lenguaje LDAP interpretado, el cual se encargará de realizar las llamadas oportunas a las funciones de generación de estructuras.

Normalmente, a cada generación de estructuras en memoria o de manejo de éstas, le corresponde un comando del lenguaje LDAP interpretado. Sin embargo, ello no siempre ocurrirá así ya que hay casos en que se usarán *operadores* del lenguaje para la llamada a las funciones de librería y casos en que diferentes comandos usarán la misma función. Los comandos de que se dispondrá en este lenguaje se pueden encontrar tanto en el ***Manual de Usuario*** como en el Cap. 6.

Como se comentó anteriormente, además del uso de cualquiera de estos comandos, el lenguaje LDAP interpretado dispone de una serie de operadores que le permitirán identificar correctamente las órdenes del usuario así como ejecutar algunas otras funciones de la librería. En especial hay dos operadores de importancia y que provocarán sendas llamadas a otras tantas funciones de la librería básica. Son los siguientes:

> Este operador señala la autoconexión entre dos pines de la célula en definición, siendo

ésta la única posibilidad de hacer uso de la función ***Def_Con()*** (ver librería ***basico.c***). Su formato de uso será el siguiente:

definición_de_pin_1>definición_de_pin_2

} Con este operador se marca el final de la definición de un elemento y, de igual forma, el final del ámbito de aplicación de cualquier comando del lenguaje (salvo SIMUL y FUNCTION, los cuales sólo se aplican sobre una línea). Sin embargo, sólo tiene su carácter de operador cuando efectivamente marcan el final de la definición de un elemento básico (célula, array, bloque o sheet) y, en cuyo caso, provocan la llamada a la función ***End_xxx()*** correspondiente al elemento cuya definición se finaliza.

En aquellos casos en que se use el operador >, éste actuará también como carácter delimitador de final de expresión para definición de pin. Es decir, cada definición de pin será analizada independientemente de las demás aunque incluya el operador >. Cuando un pin no se desee autoconectar con ningún otro de esa misma célula, su línea de definición debe finalizar con cualquier carácter delimitador que no sea > (ver ***Manual de Usuario*** para saber qué delimitadores reconoce el lenguaje LDAP interpretado).

Para trabajar con el fichero que el usuario genera con la descripción basada en el lenguaje LDAP interpretado, se ha de ejecutar, o bien el entorno ***GeneSis***, o bien el programa de análisis léxico preparado para ser usado desde la línea de comando del Sistema Operativo **UNIX**. En cualquier caso, el usuario no tiene más que indicar el nombre del fichero que contiene la descripción del sistema a generar. El analizador léxico lo tomará como su entrada estándar y comenzará a leer e interpretar su información.

Como una información errónea dada por el usuario puede abortar el proceso de generación de las estructuras de datos en memoria, habrá que prever los posibles fallos que haya en la descripción dada por el usuario, los cuales podrán ser de varios tipos:

Léxicos Alguna información dada por el usuario no se corresponde con los comandos que se esperan.

Sintácticos El formato con que se escribe un comando y sus parámetros no es reconocido por el analizador.

Semánticos Los comandos y sus parámetros están correctamente construidos, pero están siendo usados fuera de su lugar correcto.

Definición El usuario ha realizado la descripción de manera correcta en su forma, pero el sistema descrito no cumple las normas que sigue la generación de sistemas con el entorno **GeneSis**.

De estos errores, sólo el primero será detectado con el analizador generado con el LEX (no en vano es un generador de analizadores léxicos). Los otros tipos de errores habrán de ser interceptados por el propio software desarrollado bajo este analizador. Por ello, y con el objeto de evitar en lo posible la creación de estructuras de memoria incompletas, la interpretación que se hace del fichero aportado por el usuario, tiene dos fases:

Fase I Análisis de corrección del lenguaje, lo cual implica la intercepción de cualquiera de los tres primeros tipos de errores antes comentados.

Fase II Ejecución de los comandos indicados por el usuario, lo que implica la generación propiamente dicha de las estructuras en memoria.

Por supuesto, en la fase de ejecución igualmente pueden producirse errores de los que antes denominábamos de definición. En este caso se abortará inmediatamente la ejecución de las funciones y se reportará el mensaje de error adecuado. La rapidez con que se debe detener la ejecución del programa es lógica si se piensa que varios de los datos que se posicionan en memoria dependen de datos posicionados anteriormente, luego si estas operaciones previas no se han llevado a efecto debido a errores, podría darse el caso de que se produjeran un conjunto de estos en cadena y que no serán tales, sino sólo errores ficticios (consecuencia de un error anterior).

Sin embargo, cuando se detecten errores en la fase de análisis gramatical, estos no implicarán el fin inmediato de la ejecución, ya que durante esta fase no se genera ninguna estructura en memoria, sino que se espera hasta finalizar este primer paso, en cuyo momento se detendrá el proceso y no se pasará a la fase de ejecución. De esta forma, se podrán detectar la mayor cantidad de errores posibles de una sola vez.

Durante la fase de ejecución, si resulta necesario abortar la misma debido a algún error, habrá de hacerse deteniendo el análisis léxico. Para ello, y dado que éste se realiza tomando la entrada desde un fichero, bastará con situar el puntero de lectura de este fichero al final del mismo. De esta forma, cuando el analizador acuda a leer la próxima entrada se encontrará con que está al final del fichero y dará por terminada su ejecución.

3.8 Implementación.

El desarrollo de las distintas herramientas presentadas se ha llevado a cabo sobre el entorno ***OpenWindows*** debido a que ofrece una librería completa de funciones de alto nivel que permiten la creación de un entorno de trabajo sencillo y eficiente. Gracias al empleo de un entorno de programación tan difundido como lo es el ***OpenWindows***, se permitirá al usuario hacer uso, desde el propio interfase **GeneSis**, de llamadas a otras herramientas como es el caso de **SiMon**.

Todo ello, ha de desarrollarse sobre un equipo sólido en software disponible y en velocidad de proceso, con unas prestaciones generales que permitan su utilización de forma cómoda y práctica. En particular, todas las herramienta se han desarrollado sobre una estación de trabajo ***Sun-Sparc***.

La implementación se ha realizado de forma que una herramienta de carácter general pueda ser particularizada para distintos entornos de simulación. Podremos extraer de un entorno CAD la información necesaria para presentársela al usuario en la forma más oportuna.

Es importante el concepto de sucesión temporal de eventos o acciones. Los eventos los produce el sistema de simulación y las acciones las realiza el usuario sobre el sistema. Tanto las acciones como los eventos necesitan ser gestionados correctamente por el sistema.

Existen distintas formas de trabajar con el sistema:

Forma Interpretada:

Se parte de la descripción textual (*.src) y se utiliza **GeneSis** para obtener los ficheros de descripción visual (LDV) y de intercambio de variables (LIV) que serán utilizados

por el programa monitor **SiMon**. En esta opción de trabajo sólo podemos describir la arquitectura de trabajo y los vectores de cambio de variables, pero no podemos simular.

Con esta forma de trabajo podemos preparar demostraciones formativas sobre la forma de trabajo de cualquier tipo de estructura.

Forma Compilada:

En este caso a partir de una descripción textual generaremos un fichero equivalente al original con las funciones en C. Esto nos permitirá generar programas ejecutables que en función de los correspondientes parámetros de trabajo podrán generar muy distintos tipos de arquitecturas y vectores de simulación o visualización.

Forma con simulación:

En las formas anteriores solo presentamos en pantalla la arquitectura descrita y los cambios de variables que decidimos. Si además queremos simular nuestra arquitectura podemos emplear uno de los ficheros de salida de **GéneSis**. Génesis genera por defecto un fichero VHDL con la descripción de la arquitectura que se ha descrito. Esto nos permitirá simular la arquitectura descrita con cualquiera de los simuladores comerciales del mercado.

En este caso para poder ver la variación de las variables sobre SiMon, es necesario tener un interfase entre el simulador seleccionado y SiMon. Este interfase se encargará de enviarle al programa Monitor los cambios ocurridos en las distintas variables, mientras se queda a la espera de las acciones que el usuario efectúe sobre el programa monitor.

En este caso podemos tener coexistiendo ventanas con los resultados de simulación estándar y además las ventanas de SiMon.

Además de las salidas para VHDL, Génesis es capaz de generar descripciones hardware para otros lenguajes como puede ser VERILOG. En este caso para simular con VERILOG necesitamos contar con el interfase apropiado para esta herramienta.

Los objetivos generales a realizar por la herramienta serán:

* generar estructuras en memoria con la descripción del sistema,
* generar ficheros binarios donde almacenar de forma permanente la información contenida en la memoria,
* generar fichero de dibujo con formato LDV para **SiMon**, y
* generar ficheros de formato adecuado para distintos simuladores del mercado.

La generación de las estructuras de memoria habrá de poder ser llevada a cabo tanto desde el lenguaje compilado como interpretado. Los entornos con que se encontrará el usuario dependerán de la salida final que desee obtener. Para cada una de las librerías generadas,

existe un programa principal de arranque que aporta una función ***main()*** y que permite la entrada a la función principal de la librería.

3.8.1 Interfase gráfico de GeneSis.

En este apartado vamos a describir un entorno de trabajo creado para hacer uso de todas estas librerías, y proporcionar al usuario una serie de utilidades que le permitan desarrollar su trabajo de forma más cómoda y eficiente. Este entorno de trabajo recibe el nombre de **GeneSis** (*Generación de Sistemas*).

Este entorno nos permite la creación, manejo y modificación de ventanas, editores de texto, paneles de datos, y todas clase de facilidades que pueden ayudar a crear un entorno de trabajo agradable.

La ventana de la herramienta está dividida en cuatro secciones (Fig. 3.6).

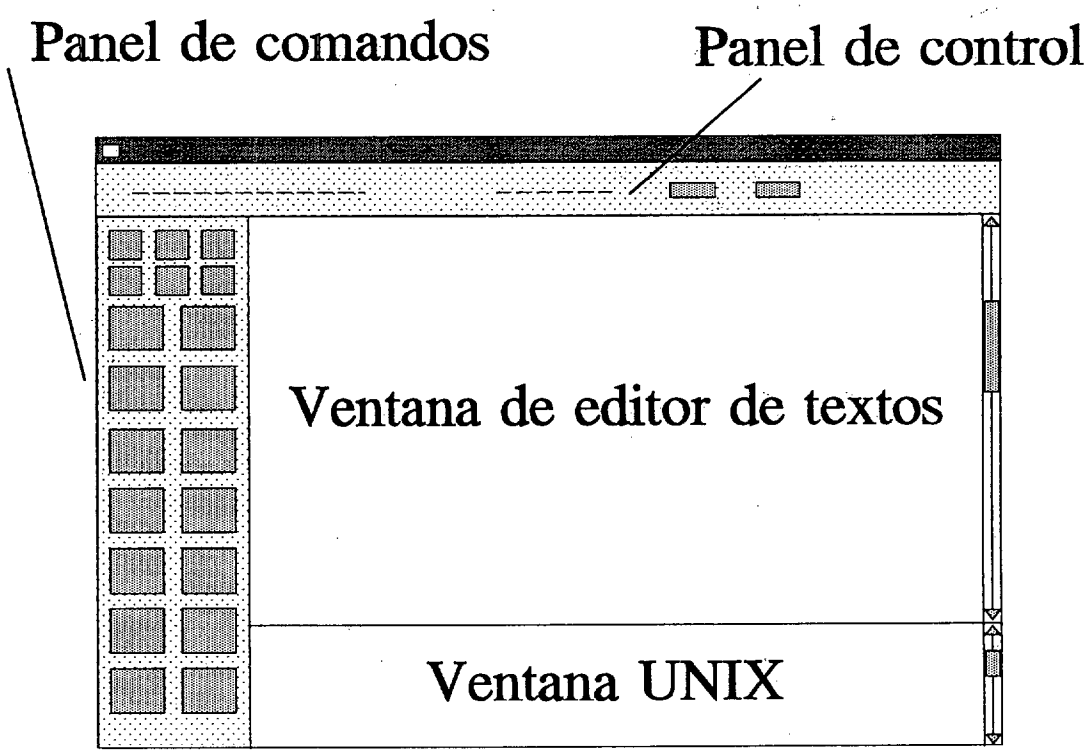

Fig. 3.6 Herramienta GeneSis.

Panel principal

En primer lugar tenemos un panel principal, en el cual podremos encontrarnos, tanto el directorio de trabajo en el que estamos actualmente, como el nombre raíz del sistema con el que vamos a trabajar. Además, también podremos encontrar facilidades para realizar la configuración de la herramienta en base a, por ejemplo, cambiar los tipos de letra con que vamos a mostrar diferentes mensajes, e incluso cambiar las extensiones que por defecto se asignan a los ficheros de salida. También tendremos utilidades para, en caso de necesidad, realizar un reinicio de todo el funcionamiento de la herramienta.

Panel de control

En segundo lugar, nos encontramos con un panel de control, donde se recoge la mayor parte de las utilidades proporcionadas al usuario. Estas se visualizan en forma de icono, cada uno de ellos asociado a un comando que se ejecutará una vez que el usuario lo seleccione a través de un *clic* con el ratón sobre el correspondiente icono. Además, no solamente disponemos de comandos asociados, sino también del camino de salida del programa mediante una función de salida indicada por un icono apropiado, así como de la posibilidad de abrir una ventana de ayuda.

Ventana de editor de textos

En tercer lugar, disponemos de un editor de texto, en la ventana central, con todas las características de los editores de texto aportados por el entorno ***OpenWindows***, con su correspondiente *scroll bar*, y todos los menús necesarios para realizar lo que es la edición de ficheros, carga, almacenaje, búsqueda de palabras, etc.

Ventana UNIX

Como cuarto punto, tenemos una ventana de comandos UNIX (en el argot de ***OpenWindows*** se denomina ventana TTY). En esta ventana tenemos un shell que se arranca automáticamente al entrar en la herramienta, es decir, al activar el programa **GeneSis**. En este shell se puede trabajar exactamente igual que si fuera una ventana de comandos del ***OpenWindows*** sobre el Sistema Operativo **UNIX**, ya que de hecho, así es, sólo que ha sido lanzada bajo control del software del programa. Por lo tanto, podremos ejecutar programas, ver listados de directorios, contenidos de ficheros, y en general cualquier comando del **UNIX**.

Estos puntos que hemos comentado aquí son los elementos visibles que tenemos en el momento de activar el interfase **GeneSis**. Cada uno de las ventanas que hemos comentado pueden tener asociado otras subventanas que, en el momento de comenzar a trabajar con el interfase, no están visibles, ya que sólo será necesaria su presencia para ciertas operaciones que el usuario desee realizar. En cualquier caso, en el momento en que sean necesarias para el funcionamiento de alguna de utilidad de las implementadas, serán abiertas de inmediato y el usuario podrá ver su contenido y/o, en su caso, modificar su valor.

La herramienta **GeneSis**, como tal, será ejecutada en una ventana del entorno ***OpenWindows***, y como tal puede ser modificada de la misma forma que cualquier otra ventana de este entorno. Puede ser abierta, puede ser cerrada, es decir, iconizada (convertida en un icono), puede ser ocultada, desplazada, etc...

Lo mismo ocurre con las subventanas que se creen desde dentro del interfase **GeneSis**, de forma que, cuando se hagan visibles por primera vez ocuparán una posición inicial determinada y fijada por programa, pero que el usuario con el ratón podrá desplazar y colocar a su conveniencia. Varias de estas subventanas, como tales, son independientes de la pantalla general de la herramienta, aunque hayan sido abiertas desde comandos situados en aquélla. Por ello, podría darse perfectamente el caso de que el usuario, por simplificar su visión de la pantalla, iconizase la ventana principal y, sin embargo, mantuviese en su estado original aquellas subventanas que hubiesen sido abiertas previamente.

Panel Principal.

En este panel se encuentran los campos y botones necesarios para que el usuario pueda

seleccionar las características de trabajo que desee tener en su entorno. Los elementos son los siguientes:

Directorio de trabajo

Indicará en qué directorio depositar los ficheros de salida y donde leer los ficheros de entrada. Para ello, dispone de una línea de entrada de datos en la que el usuario deberá escribir el nombre del nuevo directorio de trabajo.

Sólo se producirá un cambio efectivo de directorio cuando el nombre que especifique el usuario sea diferente al que había previamente.

Nombre raíz del sistema

Es el nombre raíz que se dará a todos los ficheros generados durante la sesión de trabajo. Además, cualquier fichero de entrada que desee leerse en el momento de la ejecución de alguno de los comandos formará su nombre a partir de este nombre raíz. Para ello, dispone de una línea de entrada de datos en la que el usuario habrá de introducir el nombre raíz del sistema.

Sólo se producirá un cambio efectivo del nombre raíz del sistema cuando el nombre que especifique el usuario sea diferente al que había previamente.

Botón de configuración

Permitirá configurar distintos parámetros del entorno. Tales parámetros serán los siguientes:

* tipo de letra a usar en el editor de texto,
* ficheros de salida a generar durante la ejecución del intérprete
* contenido del fichero de configuración del entorno (***genesis.setup***).

Para el fichero de configuración se permite, opcionalmente, su actualización en disco con los datos que especifique el usuario en la ventana de configuración.

Botón de reinicio

Marcado como '*CLEAR*', su efecto es el de reiniciar toda la sesión con la herramienta, de forma que se obtenga el mismo efecto que si terminase la sesión y se volviera a usar. Los cambios realizados en el fichero ***genesis.setup*** serán mantenidos.

Dado lo drástico de este comando, se solicita al usuario que confirme la operación. Si éste lo hace así, se realizarán las siguientes acciones:

* liberación de la memoria ocupada por los datos del sistema,
* borrado del nombre raíz del sistema,
* borrado del contenido del editor de texto,
* borrado del contenido de la ventana TTY de comandos, y
* cierre de todas las ventanas de apoyo que hayan podido ser abiertas por algún comando (jerarquía, conectividad ...).

Botón de Status

Muestra una información corta acerca del estado actual del sistema en memoria, reduciéndose esta información a los siguientes puntos:

* directorio actual de trabajo,
* nombre raíz del sistema, y
* si hay datos en memoria, cantidad de ésta que ocupan.

Panel de comandos.

Los comandos que proporciona el entorno se encuentran representados por unos iconos alusivos a cada uno de ellos. A continuación se da una descripción de las funciones de cada uno:

Lexer

Ejecución del intérprete, que tiene por efecto la lectura del fichero de descripción dado por el usuario, para lo cual forma su nombre a partir del nombre raíz especificado por éste en el **Panel General de Situación** añadiéndole la extensión apropiada a este fichero, y que se habrá leído del fichero de configuración ***genesis.setup***, si bien el usuario tendrá la opción de alterar esta extensión accediendo al menú de configuración.

Los mensajes de salida del análisis léxico sobre el fichero serán enviados a la ventana de mensajes genéricos de información, llamada ventana *Transcript.*

Ventana Transcript

A ella se enviarán todos los mensajes generados por la ejecución de los diferentes comandos del entorno. Puede ser abierta y cerrada cuantas veces lo desee el usuario sin que, por ello, pierda su contenido. Si se desea reiniciar éste habrá de usarse el botón '*Borrar*' que se encuentra en la cabecera de esta ventana.

La información mantenida en esta ventana, dado que se trata de un editor de texto (si bien tiene deshabilitada sus comandos de edición) puede ser trasladada a un fichero de salida, de forma que el usuario pueda mantener un registro de las acciones que ha realizado hasta entonces, durante la sesión.

Carga de datos desde disco

Lee el fichero binario con la información relativa al sistema con que el usuario desea trabajar. El nombre de este fichero se formará a partir del nombre raíz del sistema seguido de la extensión que en ese momento esté activa para este fichero.

Salva memoria en disco

Es la operación inversa a la anterior, es decir, lee el contenido del fichero binario correspondiente al sistema activo y, a partir de la información contenida en él, regenera las estructuras de datos en memoria. El nombre de este fichero se formará a partir del nombre raíz del sistema seguido de la extensión que en ese momento esté activa para este fichero.

Ejecutar programa compilado

Envía, a la ventana TTY de comandos del entorno, la orden de ejecución del programa ejecutable que haya podido generar el usuario. La ejecución de este programa tendrá efecto la generación de una serie de ficheros de salida que tendrán los nombres dados por el del sistema que haya originado esos ejecutables, pero precedido por un carácter '_'. Los mensajes que genera esta ejecución serán mostrados en la propia ventana

TTY.

Visualizador SiMon

Lanza el programa de monitorización y control **SiMon**, de forma que éste abra sus propias ventanas y organice la pantalla de acuerdo a su funcionamiento interno. La ejecución de este programa no afectará en nada al del entorno **GeneSis**.

Liberación de memoria

Libera la memoria utilizada por los datos actualmente presentes en ella, de forma que se borre todo su contenido. Dada su transcendencia, pedirá al usuario que confirme la acción a realizar, teniendo éste la opción de cancelarla.

Mostrar jerarquía

Abre la ventana de jerarquía y muestra en ésta la lista completa de elementos posicionados en todo el sistema, para cada uno de los sheets del mismo. El contenido de esta ventana podrá ser almacenado en disco si se desea, ya que se trata de un editor de texto completo.

Una vez que se ha presentado esta información, el usuario tiene la posibilidad de conseguir información adicional de cada uno de los elementos mostrados. De forma que puede pedir:

* Información descriptiva de las características de un elemento posicionado y/o su tipo (botón '*Acerca de...*').
* Información de conectividad de los pines que posee una célula posicionada (botón '*Conectado a...*').

Esta última opción provoca la apertura de una nueva ventana con la información de conectividad requerida. Por supuesto, dada las características de la información solicitada, ésta sólo será válida para elementos tipo célula y que hayan sido posicionados en el sistema, es decir, no valdrá pedir datos de conectividad acerca de tipos de elementos.

Generar fichero LDV

Genera la descripción gráfica del sistema tal y como la necesita el programa de monitorización **SiMon**, es decir, con el formato LDV. El fichero así generado puede ser recogido por éste sin necesidad de ninguna modificación adicional por parte del usuario.La extensión que se usará para este fichero será la indicada por el valor actual de la misma en el menú de configuración.

Generar fichero de información

Genera una información textual de las características del sistema descrito por el usuario. La extensión que se usará para este fichero será la indicada por el valor actual de la misma en el menú de configuración.

Compilación de LDAP

Realiza la compilación del fichero fuente con el lenguaje LDAP compilado que ha podido ser escrito por el usuario, o bien, generado automáticamente por el analizador léxico. La compilación se mandará a la ventana TTY de comandos, de forma que haga uso del fichero *gs* de procedimientos.

Carga de fichero fuente con LDAP compilado

Realiza la carga automática en el editor de texto del entorno del fichero fuente con la descripción del sistema basada en el LDAP compilado. La extensión que se usará para este fichero será la indicada por el valor actual de la misma en el menú de configuración.

Carga de fichero fuente con LDAP interpretado

Igual que en el caso anterior pero con el fichero que contiene la descripción del sistema con el lenguaje LDAP interpretado. La extensión que se usará para este fichero será la indicada por el valor actual de la misma en el menú de configuración.

3.8.2 Interfase gráfico de SiMon.

Ya se ha explicado anteriormente en este mismo capítulo que SiMon es la herramienta encargada de la monitorización y control de las distintas arquitecturas y sus variables. En este apartado vamos a ver cómo es el interfase que se presenta al usuario.

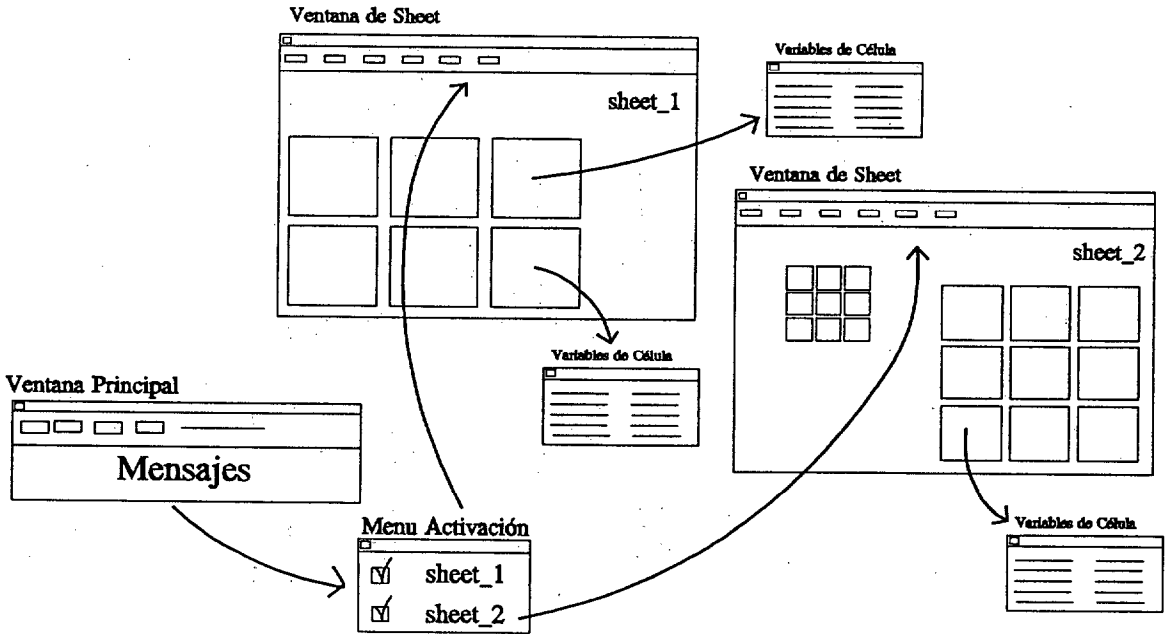

Fig. 3.7 Ventanas de SiMon.

SiMon maneja un gran número de ventanas (Fig. 3.7), que pueden ser clasificadas en dos tipos:

Ventanas del Sistema:

Son las ventanas que tenemos en número predeterminado por la arquitectura.

* **Ventana Principal**: Es la ventana que aparece al invocar la herramienta. En ella, además de otras opciones, se indicará el nombre

del fichero LDV que se quiere cargar. Posee dos paneles, uno de botones de opciones y otro de visualización de mensajes.

* **Ventana del Menú de Activación**: En esta ventana aparecerán indicados todos los Sheets que se han descrito en el fichero LDV. Desde aquí se podrán hacer visibles las distintas ventanas asociadas a cada uno de ellos.

* **Ventana de Visualización de Elementos en Memoria**: Nos permite hacer un recorrido por los distintos tipos de elementos descritos en el fichero LDV.

* **Ventanas de Sheets**: Para cada uno de los Sheets descritos tendremos una de estas ventanas que están divididas en dos zonas: una zona de comandos y otra de dibujo.

Subventanas de Sheets:

En cada una de las ventanas de Sheets podemos invocar a otras subventanas que dependen de éstas.

* Ventana de Niveles de Capas: En esta ventana podemos indicar cuales de las capas que tenemos definidas queremos visualizar en el Sheet correspondiente.

* Ventanas de Edición de Variables de Células: Para cada una de las células que tenemos posicionada en un Sheet podemos activar una ventana de edición de sus variables internas.

* Ventanas de Vistas Adicionales de la Células: A través de los comandos MODEL se pueden definir vistas externas de una célula como puede ser el caso de layout CIF.

3.9 Simulación de las arquitecturas.

En este apartado se va a explicar cómo se realiza el proceso de intercambio de información entre SiMon y cualquier otra herranienta, y cómo se realiza el proceso de simulación de las arquitecturas que ya tenemos definidas.

El entorno EASAP, es un entorno abierto que permite la posibilidad de conexión a procesos externos. El mecanismo de comunicaciión entre SiMon y el proceso que se quiere monitorizar y controlar se fundamenta en implementar un protocolo de comunicaciones a través de una zona de memoria compartida.

En esta zona de memoria compartida el proceso que quiere actualizar el valor de una variable escribirá la sentencia LIV correspondiente. En cuanto SiMon detecta que alguiente le está mandando un mensaje lo interpreta y lo ejecuta inmediatamente. El proceso inverso se produce cuando el usuario desde la pantalla actualiza el valor de una variable, que en ese mismo instante se repinta en el caso de estar visible y además se le notifica este cambio al

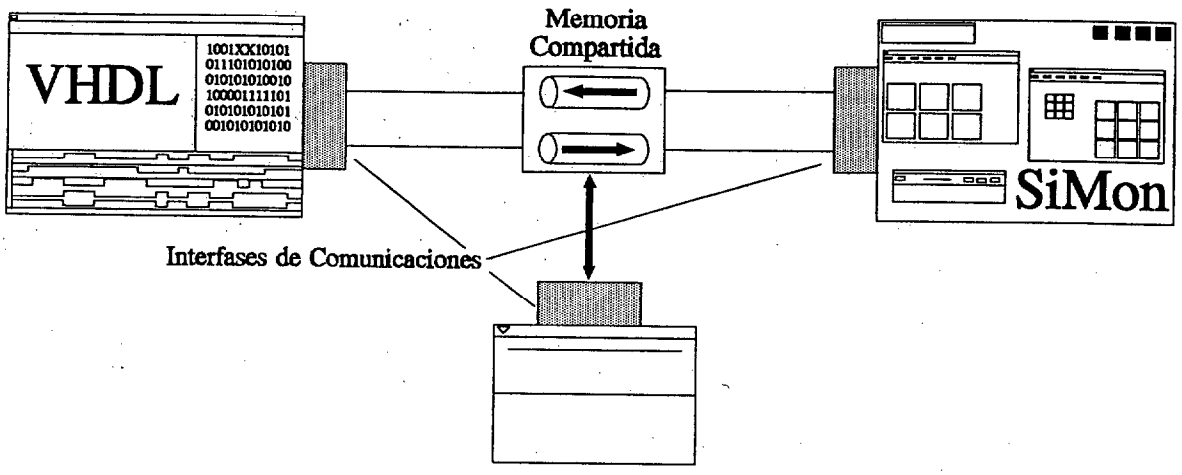

Fig. 3.8 Mecanismo de comunicación entre un simulador y SiMon.

proceso externo escribiendo el evento producido en la zona de memoria compartida para que éste lo pueda leer. Una descripción detallada de este mecanismo de comunicación se puede encontrar en el apartado 5.6.2. dedicado al interprete del lenguaje LIV.

Los simuladores digitales son un ejemplo de tipos de procesos que pueden utilizar las herramientas de EASAP como interfase con el diseñador. Para que cada usuario pueda conectar el simulador que prefiera se suministra las rutinas C que permiten manejar el protocolo descrito a través de la zona de memoria compartida.

Hay dos posibles formas de simulación:

1) La más simple es utilizar EASAP para la descripción de la arquitectura objeto de estudio. Una vez ésta se ha visualizado con SiMon y hemos comprobado que todas las conexiones se han realizado correctamente, podemos pasar a utilizar los ficheros generados por GeneSis para la simulación en un entorno comercial.

2) Una vez realizados los pasos del punto 1), se utiliza SiMon para visualizar los datos del simulador.

Capítulo 4.

Ejemplos de uso del entorno EASAP.

4.1 Estudio de métodos de multiplicación de vectores y matrices.

Existen distintos campos en los que se requiere el cómputo reiterativo de multiplicaciones de vectores y matrices, pero quizás el campo que realiza un uso más exhaustivo es el del procesado de señal.

El procesado de señal abarca una gran variedad de técnicas que van desde el acondicionamiento de la señal "signal conditioning" en el bajo nivel, hasta la interpretación de la señal en alto nivel. Las técnicas de acondicionamiento de la señal como el filtrado, pretende obtener una vista más útil de los datos sin procesar. En el nivel intermedio tenemos operaciones descriptivas que pretender reducir los datos y pasarlos a una forma más abstracta y en el nivel más alto las señales pueden ser clasificadas o interpretadas para determinar algún tipo de acción.

Muchas de las operaciones de tratamiento de la señal, especialmente aquellas en niveles bajos o intermedios, están basadas en la multiplicación de vectores de datos por matrices o vectores de coeficientes. Por ejemplo, en los filtros FIR o IIR, las salidas está obtenida por la multiplicación de un vector de datos por otro de coeficientes. Los bancos de filtros son ampliamente utilizados para extracción de características, por ejemplo en los canales de voz. En el nivel más alto, lo métodos de reconocimiento de patrones pueden ser usados para clasificación de vectores característicos, con lo que de nuevo aparece el problema de la multiplicación vector-matriz.

En el procesamiento de señal en tiempo real se requiere el uso frecuente de métodos de computación paralelos y "pipelined". Estas ideas son inherentes a los arrays sistólicos, los cuales obtienen los resultados mediante iteraciones de cadenas de datos.

En este punto se pretende dar una visión general del problema de encontrar una array sistólico eficiente para resolver la multiplicación de vectores y matrices. La exposición tendrá dos grandes bloques, o tipos de arrays, los que trabajan a nivel de palabra y los que lo hacen a nivel de bit. En los primeros de éstos, cada una de las células de los arrays operan con datos con un ancho de palabra determinado, lo que implica que en realidad los canales o puertos de comunicación entre células adyacentes son buses. En cambio en los arrays que operan con bits las comunicaciones se realizan con líneas simples de conexión.

4.1.1 Arrays sistólicos a nivel de palabra.

Antecedentes

La mayoría de los arrays sistólicos que fueron investigados inicialmente, utilizaban operaciones a nivel de palabra como primitivas. Uno de los problemas más simples en los que fueron empleados fue la multiplicación de matrices y vectores.

Procesador de paso de producto interno.

Una gran parte de los algoritmos que vamos a considerar a continuación, utilizan como operación más simple, común a todos ellos, el llamado paso de producto interno "inner product", $C <- C + A \times B$. La célula básica o procesador que realiza esta operación está compuesto de tres registros R_A, R_B y R_C (Fig. 4.1). Cada registro está controlado por la señal de reloj que hace que en cada ciclo de reloj, los datos se desplacen de las entradas de los registros a las salidas de éstos, computando la operación $R_C <- R_C + R_A \times R_B$. Como las salidas de los procesadores están conectadas con las entradas de los vecinos, podemos utilizar registros de forma que el cambio de la salida de un registro durante una unidad de tiempo no interfiera con la entrada de otro durante ese mismo intervalo de tiempo. Las geometría de estos procesadores básicos de productos interno pueden ser rectangulares o hexagonales en función del tipo de array en que operen.

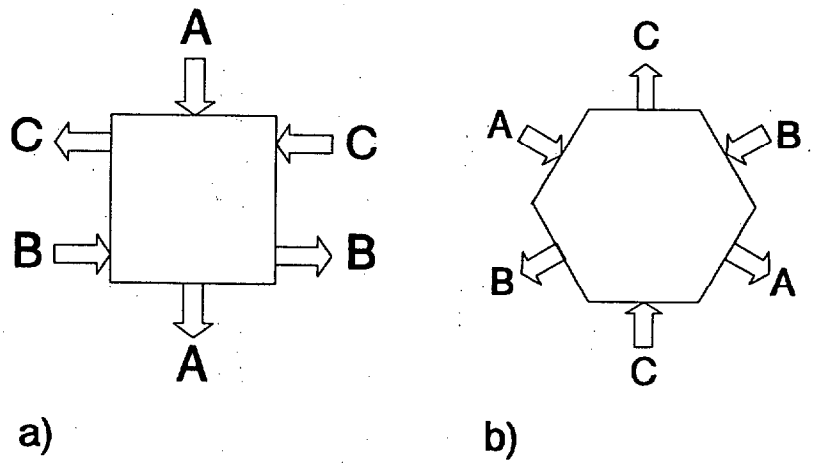

Fig. 4.1 Geometrías de un procesador de producto interno.

Multiplicación de matriz completa por vector.

Supongamos que queremos multiplicar una matriz W de n x m por un vector columna X de m elementos para dar un vector resultante G de n elementos. Los elementos del producto G pueden ser calculados por las siguientes recurrencias:

$$g_i^{(1)} = 0,$$

$$g_i^{(k+1)} = g_i^{(k)} + w_{ik}x_k,$$

$$q_i = g_i^{(n+1)},$$

$$\begin{bmatrix} w_{11} & w_{12} & \dots & w_1m \\ w_{21} & w_{22} & \dots & w_2m \\ . & . & \dots & . \\ w_{n1} & w_{n2} & \dots & w_{nm} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ . \\ x_m \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ . \\ g_n \end{bmatrix}$$

Para realizar esta operación podemos emplear una array lineal, cuyo flujo de datos circule en un solo sentido. Primero tendremos que almacenar los elementos del vector X en cada una de las células del array. La operatividad de esta estructura se consigue pasando entonces los elementos de la matriz W a través del array (Fig. 4.2). La cadena de datos W se introducirá en el array lineal desde la derecha e interactuará con el vector X almacenado para obtener una cadena resultante de salida G, cuyo flujo de datos circulará desde arriba hacia abajo. Como se puede observar necesitamos para esta operación un array lineal de células de procesamiento de longitud igual al del vector X. En la Fig. 4.2 podemos ver un ejemplo en donde trabajamos con un matriz W de 4 x 5 y con un vector X de 4.

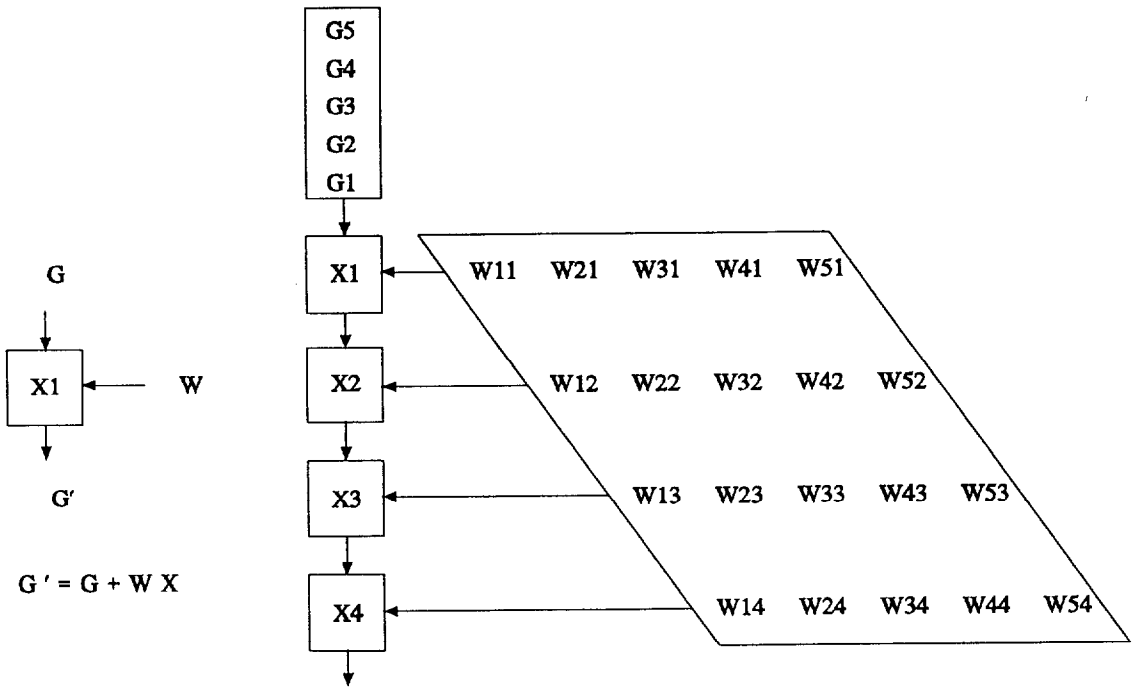

Fig. 4.2 Multiplicación matriz-vector 1.

Multiplicación de matriz banda por vector

Si W es una matriz banda de n x m con un ancho de banda de $w = p + q - 1$, entonces existe ventaja en la utilización de un array lineal de dos sentidos, en el que ahora el vector de X de entrada atraviesa el array en sentido ascendente mientras que el vector de G resultados, que inicialmente es cero, fluye en sentido opuesto (Fig. 4.3 en donde $p=2$ y $q=3$). Al igual que en el caso anterior, los coeficientes se introducen desde la derecha, pero en este caso la secuencia varía de la anterior. Todos los movimientos están sincronizados.

$$\begin{bmatrix} w_{11} & w_{12} & 0 & 0 & 0 & 0 & . & . \\ w_{21} & w_{22} & w_{23} & 0 & 0 & 0 & . & . \\ w_{31} & w_{32} & w_{33} & w_{34} & 0 & 0 & . & . \\ 0 & w_{42} & w_{43} & w_{44} & w_{45} & 0 & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_m \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ . \\ . \\ . \\ g_n \end{bmatrix}$$

De cualquier manera la introducción de flujos en sentidos contrarios obliga a la introducción de ceros entre los datos, haciendo que la eficacia del array sea solo del 50%. Adviértase que el tamaño del array está determinado en este caso por el ancho de banda de la matriz W, en lugar de por su número de filas.

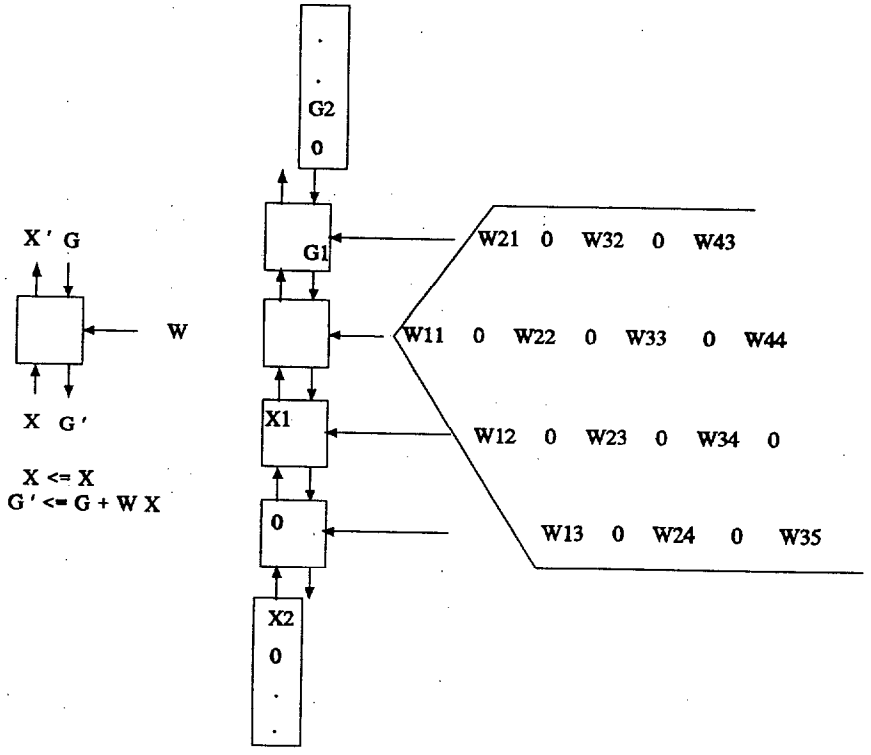

Fig. 4.3 Multiplicación matriz-vector 2.

Para explicar el algoritmo más claramente, vamos asumir que todos los procesadores están numerados de 1 a w, comenzando a numerarlos de abajo a arriba, en donde w es el ancho de

banda de la matriz W (Fig. 4.4). Cada uno de los procesadores tiene tres registros, R_w, R_x, y R_g, los cuales mantienen respectivamente los datos de W, X y G. Inicialmente todos los registros están inicializados a cero. Es de destacar que en cada paso de tiempo impar del proceso únicamente estarán activadas las células cuya numeración sea impar, y en los paso de proceso pares estarán activadas las células de numeración par. El proceso a seguir es el siguiente:

Paso 0: Inicialmente todos los registros de las células están inicializados a cero. En este momento g_1 se introduce en el procesador 4.

Paso 1: x_1 es introducido en el procesador 1, mientras g_1 se ha desplazado un lugar hacia abajo.

Paso 2: w_{11} es introducido en el procesador 2 a la vez que se desplazan x_1 y g_1. En este momento w_{11}, x_1 y g_1 coinciden en el procesador 2. Simultáneamente se ha introducido g_2 en el procesador 4. La operación que se realiza es:

$$g_1 = w_{11}\, x_1$$

Paso 3: w_{12} y w_{21} se introducen simultáneamente en el array en los procesadores 1 y 3 respectivamente. Se realizan en este caso dos operaciones:

$$g_1 = w_{11}\, x_1 + w_{22}\, x_2$$
$$g_2 = w_{21}\, x_1$$

Paso 4: w_{22} y w_{31} se introducen simultáneamente en el array en los procesadores 2 y 4 respectivamente. En este paso g_1 sale del array. Las operaciones que se realizan son:

$$g_2 = w_{21}\, x_1 + w_{22}\, x_2$$
$$g_3 = w_{31}\, x_1$$

Paso 5: w_{23} y w_{32} se introducen simultáneamente en el array en los procesadores 1 y 3 respectivamente. Las operaciones son:

$$g_2 = w_{21}\, x_1 + w_{22}\, x_2 + w_{23}\, x_3$$
$$g_3 = w_{31}\, x_1 + w_{32}\, x_2$$

Paso 6: w_{33} y w_{42} se introducen simultáneamente en el array en los procesadores 2 y 4 respectivamente. En este paso g_2 sale del array. Las operaciones que se realizan son:

$$g_3 = w_{31}\, x_1 + w_{32}\, x_2 + w_{33}\, x_3$$
$$g_4 = w_{42}\, x_2$$

Usando un procesador de producto interno, podemos observar que las tres operaciones de desplazamiento que se realizan en el paso tres, pueden ser efectuadas simultáneamente. Si el ancho de banda de W es w, podemos observar que después de w ciclos de reloj los componentes del producto $g=Wx$ comienzan a desplazarse fuera del último procesador con

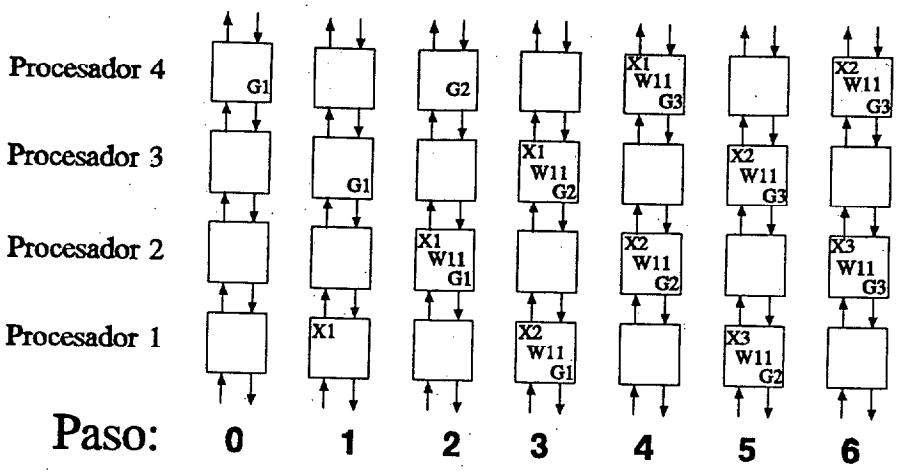

Fig. 4.4 Pasos de trabajo.

una frecuencia de cada dos ciclos. Usando esta aproximación todos los términos de g pueden ser obtenidos en un tiempo $2n + w$, en comparación con $O(wn)$ tiempos necesitados por un algoritmo secuencial sobre un sistema uniprocesador.

Un ejemplo del uso de multiplicaciones de vector por matriz de banda en procesamiento de señal lo tenemos en el filtro FIR en donde los coeficientes del filtro están representados por una "upper triangular-band Toeplitz-matrix". Esto significa que los coeficientes que se introducen en un elemento de procesamiento determinado del array tienen el mismo valor y podrían ser almacenados en el array.

Multiplicación matriz-matriz

Existen otras aplicaciones en las que es necesario transformar una secuencia de segmentos de señal, entonces podemos representarla como una multiplicación matriz-matriz.

Consideremos el problema de multiplicar una matriz $A=(a_{ik})$ $q \times r$ por una matriz $B=(b_{kj})$ $p \times q$ para obtener una matriz $C=(c_{ij})$ $p \times r$.

$$A=(a_{ik})$$
$$B=(b_{kj})$$
$$C=BA=(c_{ij})$$

$$c_{ij}=\sum_{k=1}^{q} b_{ik}a_{kj}$$

Hwang y Cheng [HwCh80] describen un array rectangular para la multiplicación de matriz-matriz en el cual mueven las matrices de entrada en sentidos opuestos para formar una región de interacción en forma de paralelogramo (Fig. 4.5). Adviértase que la región de interacción viaja ortogonalmente al flujo de datos de entrada. El tamaño del array es $q \times (p + r - 1)$ y la eficacia tiende hacia el 25% como p y q crecen (asumiendo $p=r$). El número de procesadores inactivos se incrementa con el hecho de que las matrices de entrada contienen ceros

interpuestos y debido también a la existencia de procesadores inactivos entre regiones que están interactuando.

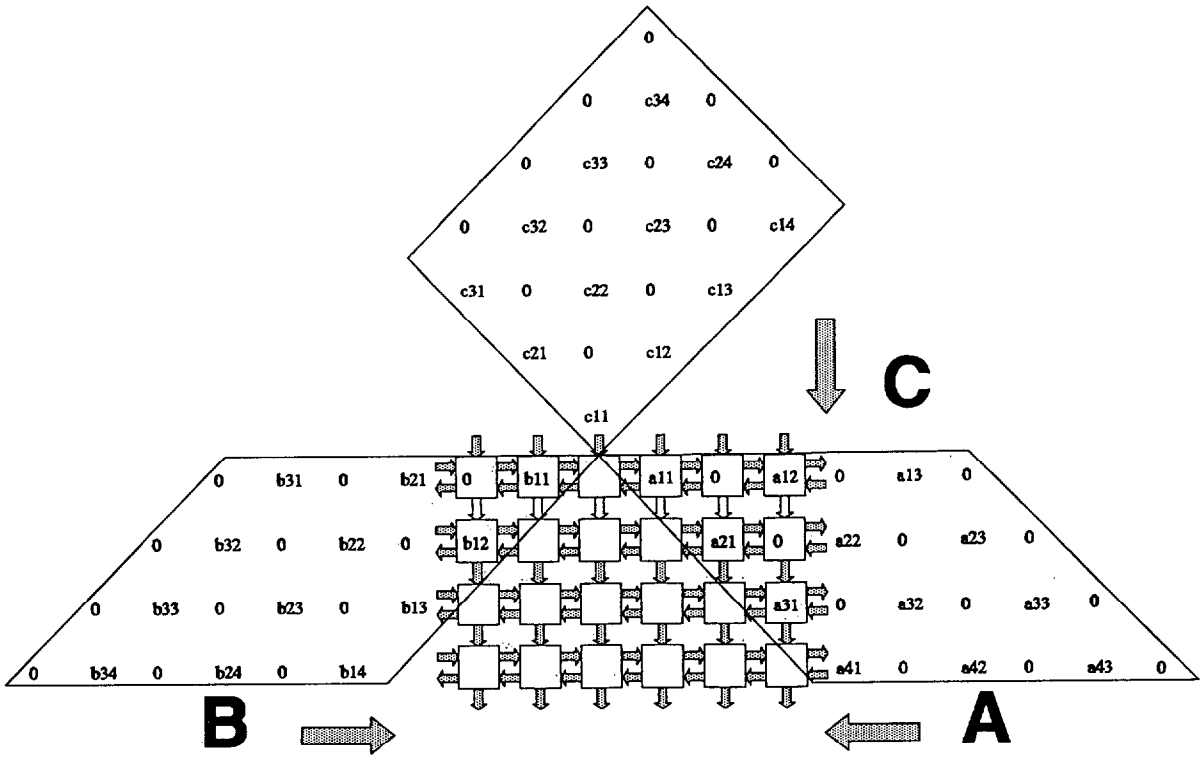

Fig. 4.5 Multiplicación matriz-matriz.

Mejora de la eficiencia:

Fijemos ahora nuestra atención en los métodos de mejora de la eficiencia en arrays sistólicos. Nos concentraremos en el multiplicaciones matriz-matriz, pues partimos del conocimiento de que en las multiplicaciones matriz-vector podemos llegar a alcanzar una eficiencia del 100%. Está claro que la falta de eficiencia radica en la necesidad de intercalar ceros en la cadena de datos de entrada y en la sucesiva interacción de regiones "not butting".

Es significativo señalar que en tratamiento de señal, la mayoría de las multiplicaciones requieren el empleo de coeficientes fijos.

Volviendo al algoritmo de multiplicación de matrices, hay que señalar que si B es la matriz de coeficientes entonces la multiplicación puede realizarse en un array de tamaño $q \times r$.

4.1.2 Arrays sistólicos a nivel de bit.

Cuando estamos trabajando con arrays sistólicos a nivel de palabra, existe un serio problema al incrementarse el ancho de banda de las entradas-salidas cuando intentamos integrar los elementos de proceso en un único chip, lo que suele ser impracticable. En este punto hay que resaltar que las ventajas que teníamos al utilizar arquitecturas paralelas desaparecerán en gran medida si tenemos que utilizar comunicaciones series a nivel de bit entre procesadores.

Una solución es utilizar aritmética "bit serial" y comunicaciones "bit-serial word-parallel" [PoWe80].

> Los primeros arrays sistólicos que empleaban operaciones a nivel de bit fueron descritos por Foster y Kung [FoKu80]. Sus aplicaciones requieren idéntica interacción entre los datos, aunque se utilizaban bits en lugar de palabras.
>
> McCanny y McWhirter [CaWh82], hicieron desarrollos más interesantes en donde incluían arrays trabajando con cadenas de datos "bit parallel" como en "bit serial".

Los arrays "bit-level" son además de considerable interés para las operaciones de multiplicación de vectores y matrices, por su compatibilidad con "bit-serial word-parallel I/O". Es conocido que existe una fuerte conexión entre operaciones de multiplicación a nivel de palabra y a nivel de bit. Podemos utilizar estructuras similares para operaciones de multiplicaciones de matrices a nivel de palabra y para productos internos a nivel de bit. Existe además relación entre el producto interno en el nivel de palabra y multiplicaciones en el nivel de bit. Algunas operaciones pueden ser realizadas usando arrays sistólicos con idénticos flujos de datos que sus arrays análogos a nivel de palabra.

Los arrays a nivel de bit descritos en [WhCa82] y [CaWo83] usan un esquema de producto interno que es esencialmente análogo al array de Hwang y Cheng [HwCh82]. A diferencia de la convolución sistólica a nivel de palabra que se presentaba en la Fig. 4.3 no es necesario introducir palabras nulas (ceros) en las cadenas de entrada y salida. En lugar de eso se introducen bit ceros entre los datos. Cada fila del array descrito en [WhCa82] es básicamente un multiplicador serie paralelo, pero a diferencia de los multiplicadores serie paralelo, los productos parciales se propagan a través del array y son acumulados en el array de borde.

Los arrays a nivel de bit análogos a los descritos por Hwang y Cheng [HwCh82] tiene los mismos problemas de eficiencia. Se han desarrollado un gran número de métodos [Ur83][CoPa83][CaWh83] para conseguir incrementar la eficacia de estos arrays hasta el 50% eliminando los ceros interpuestos, pero estas soluciones dejan regiones de procesadores inactivos entre sucesivas regiones de interacción.

Mejora de la eficiencia de los arrays a nivel de bit

Para la mejora de eficiencia de los arrays a nivel de bit, vamos a realizar un estudio detallado del uso de coeficientes estáticos en lugar de coeficientes de carga dinámicos. Partiremos de la definición de una serie de términos para la formación de un producto G a partir de una palabra de datos X y una palabra de coeficientes W, en donde X y W tienen respectivamente B y C bits.

$$X=(x^{(B-1)}x^{(B-2)},\ldots,x^{(1)}x^{(0)})$$
$$W=(w^{(C-1)}w^{(C-2)},\ldots,w^{(1)}w^{(0)})$$
$$G=(g^{(B+C-1)},\ldots,g^{(1)}g^{(0)})$$

Teniendo en cuenta lo anterior, el producto G puede ser formado mediante la suma de los

productos parciales, los cuales se definen de la siguiente manera:

$$g_{ij}=x^{(i)}w^{(j)}$$

En la Fig. 4.6 tenemos un ejemplo del cálculo del producto G a partir de un dato con $B=4$ y coeficiente con $C=4$.

$$\begin{array}{cccccccc}
 & & & x^{(3)} & x^{(2)} & x^{(1)} & x^{(0)} & \\
 & & & w^{(3)} & w^{(2)} & w^{(1)} & w^{(0)} & \\
\hline
 & & & g_{30} & g_{20} & g_{10} & g_{00} & \\
 & & g_{31} & g_{21} & g_{11} & g_{01} & & \\
 & g_{32} & g_{22} & g_{12} & g_{02} & & & \\
g_{33} & g_{23} & g_{13} & g_{03} & & & & \\
\hline
g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0
\end{array}$$

Fig. 4.6 Multiplicación de dos palabras de 4 bit.

Si almacenamos los bit de W a lo largo de una cadena de células C, podemos formar un paralelogramo de productos parciales. Para eso debemos introducir los bits de la palabra X desde la derecha en el array de células C. Se introducen los bits de X comenzando por el de menor peso en el primer instante. El resto de los bits se introducen en los siguientes ciclos de reloj. Como cada célula desplaza el bit de X que le llega desde la derecha hacia la izquierda a través de un elemento de un registro de retardo, la cadena de bit atraviesa todo el array. Si el bit menos significativo de cada palabra interactúa primero, los productos parciales de igual peso emergerán desde la cadena simultáneamente tal como podemos ver en la Fig. 4.7.

Por el contrario si los bits de la palabra W están introducidos de forma inversa tal como vemos en Fig. 4.8 los productos parciales de igual peso aparecerán en el paralelogramo de productos parciales en diagonal.

Tanto en la estructura de la Fig. 4.7 como en la estructura de la Fig. 4.8, si una secuencia continua de palabras X están multiplicadas por W, sus paralelogramos de productos parciales se irán generando sin necesidad de intercalar ceros entre las sucesivas palabras, con lo resultará que tendremos una utilización del 100% de los elementos procesadores.

Tomando como base las estructuras anteriores, podemos realizar una ampliación que nos permita la formación de productos internos G^* de vector a partir de X y de W.

N multiplicadores del tipo descrito arriba pueden estar en cascada para dar un array bit level

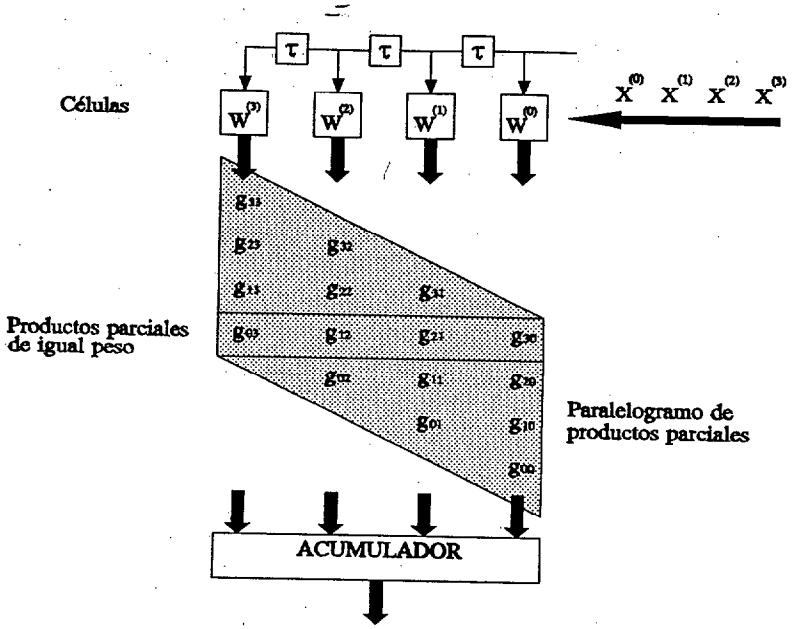

Fig. 4.7 Formación de un paralelogramo de productos parciales.

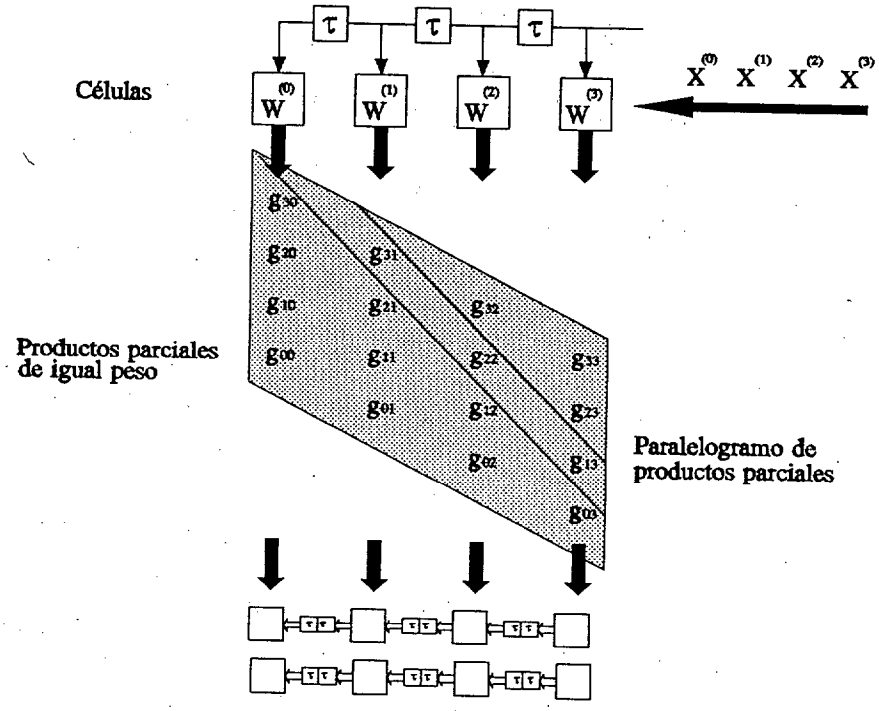

Fig. 4.8 Formación alternativa de productos parciales.

análogo al de la Fig. 4.5. En este caso el vector de entrada está desplazado e introducido en el array a partir de su bit menos significativo (Fig. 4.9). Cada una de las células del array principal está compuesta por un sumador (full-adder) y por cuatro registros para los datos positivos y un full-adder y cinco registros por datos en complemento dos (Fig. 4.10). El

$$X = (X_1, X_2, \ldots, X_N)$$
$$W = (W_1, W_2, \ldots, W_N)$$

$$G^* = \sum_{k=1}^{N} X_k W_k$$

paralelogramo de productos parciales se mueve hacia abajo a través del array acumulando su contribución para el cálculo del producto interno. Se podrá observar que el acarreo se encuentra recirculando. El crecimiento de la palabra está permitido si añadimos una banda de ceros (*log2 N*) al final de cada vector de entrada.

El producto interno G^* es obtenido mediante la acumulación de los términos del paralelogramo. El método para acumular los productos parciales tal como estudiamos en la Fig. 4.7 puede ser mediante la utilización de una cadena lineal de células de un "carry save adder", o mediante el empleo de un acumulador de árbol (Fig. 4.8):

a) desplazando (deskewing) el paralelogramo y acumulando a lo largo de una cadena lineal de células con un "carry save adder".

b) utilizando un árbol de sumadores.

El paralelogramo de la Fig. 4.7 puede ser acumulado usando una cadena lineal de células acumuladoras. Cada una de las células acumuladoras se componen de un full-adder y cuatro registros.

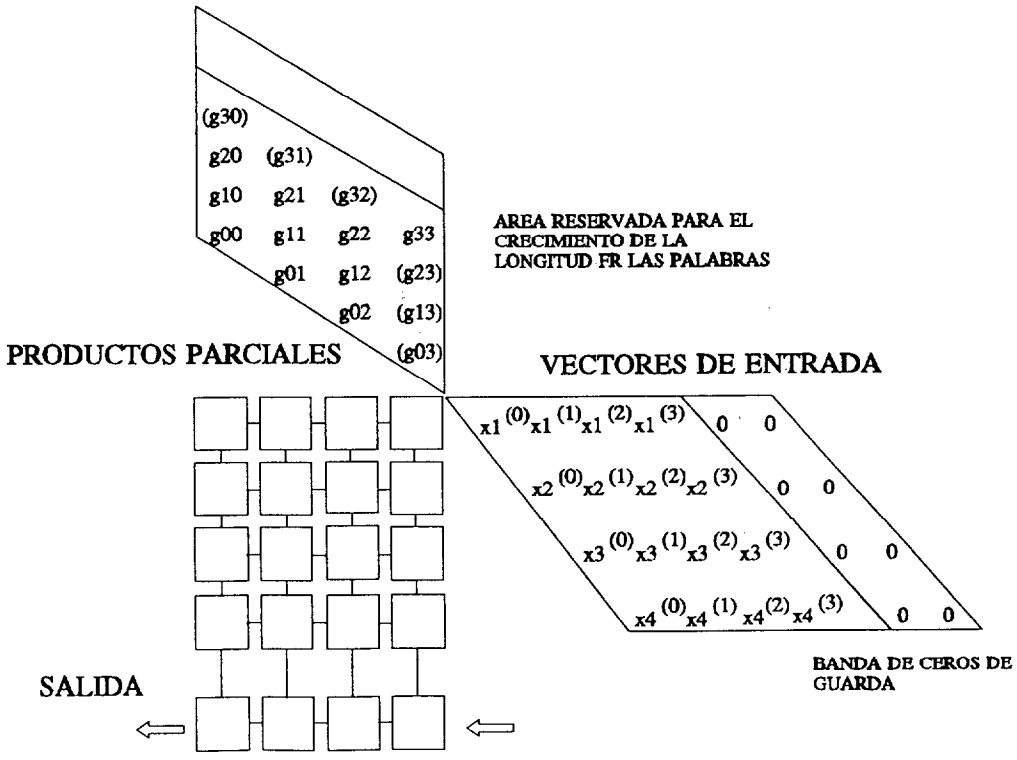

Fig. 4.9 Formación de un producto interno.

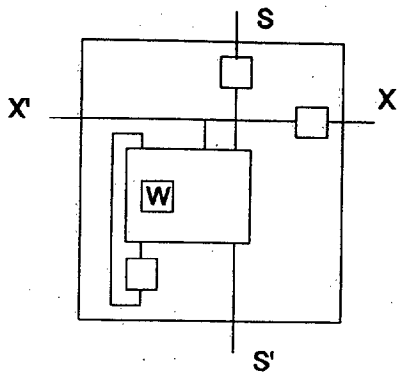

Fig. 4.10 Célula principal del array.

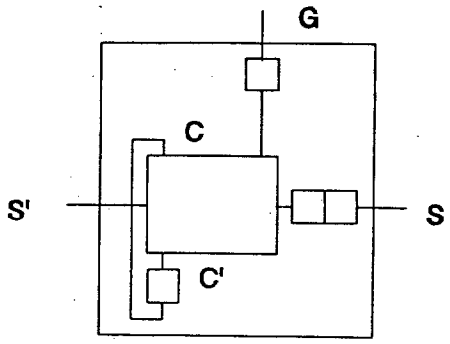

Fig. 4.11 Célula del acumulador.

Si nos fijamos en la Fig. 4.7 queda patente que la velocidad de G^* es justo la mitad que la de los datos de entrada; en otras palabras, un bit tarda el doble de tiempo en moverse a través de las células del acumulador que entre las células del array. Esto significa que si usamos un simple acumulador, los datos todavía estarán en el array cuando el próximo paralelogramo llegue. Esto significa que de no usar la estrategia expuesta sería necesario introducir C ceros a la banda de guarda o bien necesitaríamos dos acumuladores trabajando en las regiones de interacción alternativas.

Como resultado tenemos una eficiencia en el array del producto interno de valor $B/(B+\log_2 N)$ para los arrays de doble acumulador y de $B/(B + \log_2 N + C)$ para los arrays de simple acumulador. Estas cifras de eficiencia sólo toman en consideración la generación de productos parciales, en el array de doble acumulador todas las células que no se encuentran activas en la generación de los productos internos están activas acomodando el crecimiento del ancho de palabra. Si existen variaciones en el número de bits de ancho de las palabras de entrada, no es necesario modificar el tamaño del array, únicamente el ancho de la región de interacción.

Aritmética en complemento dos

Hasta ahora hemos considerado las operaciones con aritmética binaria positiva. La mayoría de las aplicaciones requieren operar con números negativos, por lo tanto necesitamos adaptar el array de producto interno que hemos estudiado para permitirle trabajar con aritmética en complemento dos. Recordamos que en un número en complemento dos el bit más significativo indica el signo. Este bit vamos a resaltarlo presentándolo entre paréntesis.

En el caso de la multiplicación de X por W, si ambos números están en complemento dos, se generarán los productos parciales positivos y negativos. Por conveniencia seguiremos el algoritmo de Baugh-Wooley [BaWo83] y separaremos los términos positivos y negativos.

En un array sistólico es deseable usar los productos parciales existentes y aplicar una operación simple a los términos de peso negativo. Se puede invertir los términos de peso negativo que se muestran entre paréntesis en la Fig. 4.9 y añadir una ajuste aritmético el cual corrija el producto final [BaWo83][CaWh84]. Este ajuste aritmético está fijado en función del ancho de palabra que se utilice. Consideremos la multiplicación de dos números en complemento dos de la siguiente forma:

$$X=((x^{(B-1)})x^{(B-2)},\ldots,x^{(1)}x^{(0)})$$
$$X=-x^{(B-1)}2^{B-1}+\sum_{k=0}^{B-2} x^{(k)}2^k$$

$$W=((w^{(C-1)})w^{(C-2)},\ldots,w^{(1)}w^{(0)})$$
$$W=-w^{(C-1)}2^{C-1}+\sum_{k=0}^{C-2} w^{(k)}2^k$$

$$G=(g^{(B+C-1)},\ldots,g^{(1)}g^{(0)})$$

$$W = -w_{n-1}2^{n-1} + \tilde{w}$$
$$X = -x_{n-1}2^{n-1} + \tilde{x}$$

en donde $w^{\sim}$ y $x^{\sim}$ son los términos compuestos por los ***n-1*** bit menos significativos de W y X. El producto de estos dos números está claramente expresado en la siguiente ecuación:

$$W.X = w_{n-1}x_{n-1}2^{2n-2} + \tilde{w}\,\tilde{x} - \tilde{w}x_{n-1}2^{n-1} - \tilde{x}w_{n-1}2^{n-1}$$

y esta expresión puede ser escrita en forma que envuelva solo los productos parciales positivos y resaltando que los términos como

$$-\tilde{w}x_{n-1}$$

pueden ser escritos de la forma

$$-\overline{w}x_{n-1}=\left(\overline{\overline{w}x_{n-1}}+1-2^{n-1}\right)$$

la ecuación Ecu(4.10) podemos por lo tanto expresarla como

$$W.X = \left(w_{n-1}x_{n-1}\right)2^{2n-2} + \tilde{w}\tilde{x} + \left(\overline{\tilde{w}xn-1} + \overline{\tilde{x}w_{n-1}}\right)2^{n-1} + 2^n - 2^{2n-1}$$

En las ecuaciones anteriores puede verse que la multiplicación de dos números en complemento dos puede ser escrita en forma que envuelva solo productos parciales positivos con tal que todos los productos parciales que envuelven bit con signo y sin signo estén complementados. La respuesta se obtiene entonces añadiendo un término de corrección fijo al resultado final.

La aplicación de la ecuación Ecu(4.13) para el caso de 4 bit en X y W se ilustra en la Fig. 4.12 en donde podemos comprobar que la adición del término -2^{2n-1} se ha tenido en consideración con la adición de un 1 a la columna más significativa del resultado y el término 2^n con la adición de un 1 en la columna (n+1).

$$
\begin{array}{cccccccc}
 & & & & (x_3) & x_2 & x_1 & x_0 \\
 & & & & (w_3) & w_2 & w_1 & w_0 \\
\hline
 & & & & \overline{g_{30}} & g_{20} & g_{10} & g_{00} \\
 & & & \overline{g_{31}} & g_{21} & g_{11} & g_{01} & \\
 & & \overline{g_{32}} & g_{22} & g_{12} & g_{02} & & \\
 & g_{33} & \overline{g_{23}} & \overline{g_{13}} & \overline{g_{03}} & & & \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\hline
g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0
\end{array}
$$

Fig. 4.12 Multiplicación de dos palabras de 4 bit en complemento dos.

En un array la inversión del producto parcial está controlada mediante la propagación de un patrón de señal de control (*ctrl*) a través del array que indique a cada célula en cada momento si necesita realizar la inversión o no. Normalmente este patrón es una cadena de bit que cuando su valor esta a "1" indica que se ha de realizar la inversión [WoEv83][UrWo84][CaWh84]. Esto se consigue añadiendo un registro que desplace verticalmente la señal *ctrl* (Fig. 4.13). La adición del término apropiado de corrección puede ser manejado de manera simple inicializando adecuadamente el acumulador del array al valor adecuado.

Carga de coeficientes

Un problema que hay que tener en cuenta es la carga de los coeficientes. Dado un array determinado, normalmente es necesario realizar operaciones de multiplicaciones de matrices y vectores de dimensiones inferiores al tamaño del array. En estos casos, hay que prever algún método de carga rápido de los coeficientes.

En los arrays, sólo las señales de alimentación y el reloj suelen ser líneas de distribución global. La transmisión de datos se realiza de forma local entre las células. De todas formas los datos pueden ser cargados rápidamente utilizando un bus de datos que corra a lo largo del array, al cual sólo existirá una célula conectada a la vez.

El mecanismo de carga de estos coeficientes, consiste en propagar un patrón de bits de carga (*load*) a través del array. Los bits de los coeficientes se introducen en las líneas del bus de datos. Como estas línea corren a lo largo de todo el array en el sentido vertical, según vamos introduciendo los bits, éstos llegan a todas las células de la misma columna. La señal *load*

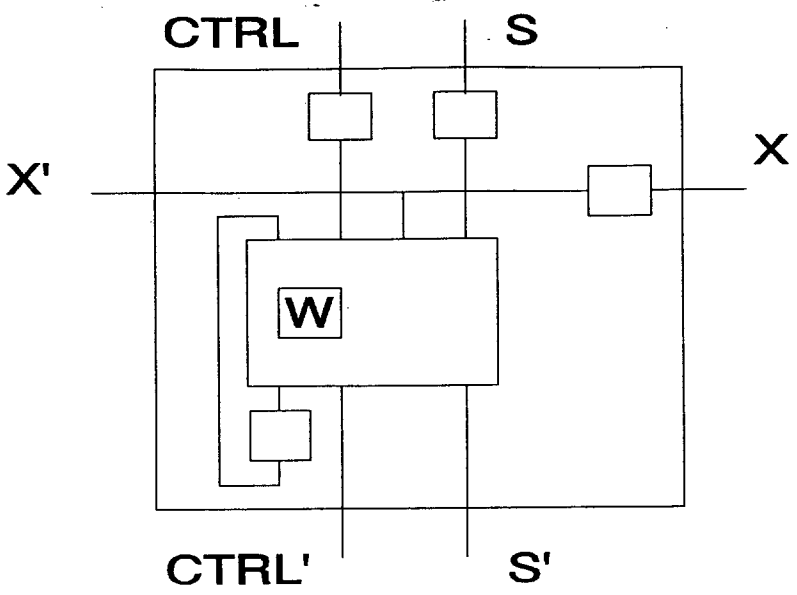

Fig. 4.13 Interior de una célula con mecanismo de control para trabajar en complemento 2.

consiste en un uno seguido de una cadena de ceros, en la Fig. 4.14 podemos ver como la señal ***load*** se propaga de célula en célula en cada ciclo de reloj. El uno se irá desplazando en cada columna de arriba a bajo, y en cada célula en la que el uno esté presente se cargará el bit del coeficiente que esté presente en ese momento el la línea del bus.

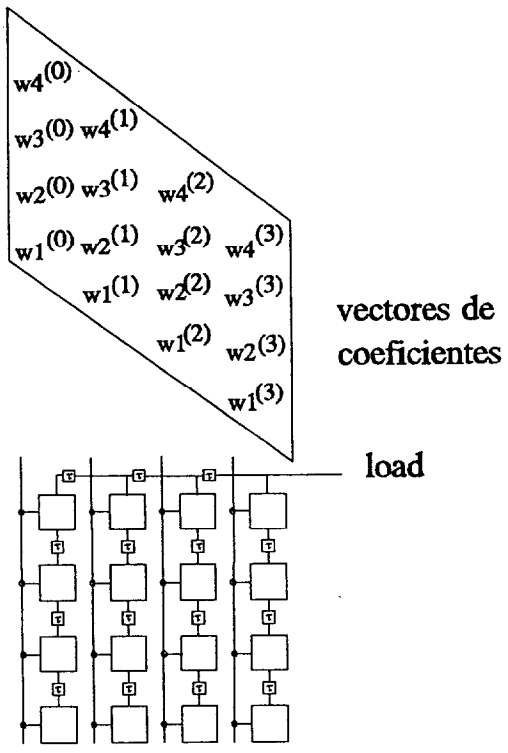

Fig. 4.14 Mecanismo de carga de coeficientes mediante la señal LOAD.

Con este mecanismo conseguimos una carga rápida de los bit de los coeficientes en el array, y además sólo necesitamos incrementar el área ocupada por cada célula un pequeño porcentaje (Fig. 4.15).

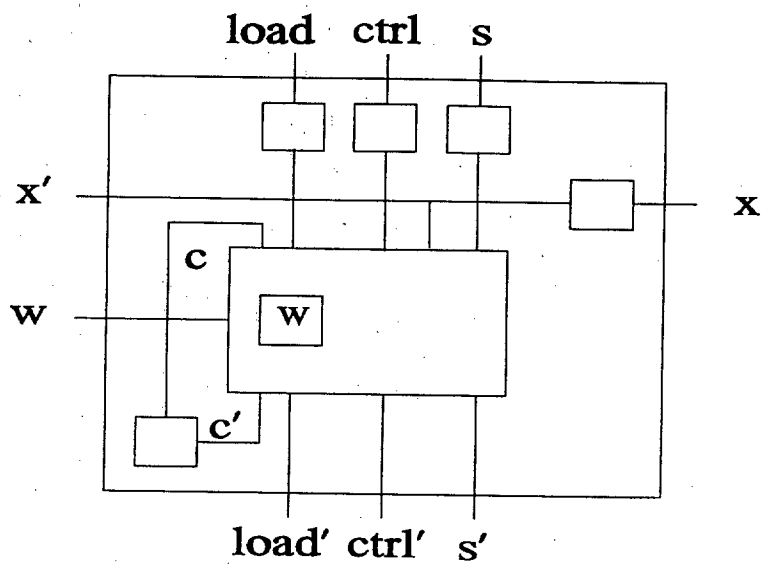

Fig. 4.15 Interior de una célula con su mecanismo de carga de coeficientes.

Métodos de mejorar la latencia.

En muchas de las aplicaciones, en donde este tipo de estructuras son utilizadas, es necesario tener una rápida respuesta ya que no se pueden tolerar tiempos de latencia elevados. Una forma evidente de mejorar la latencia de un array de producto interno, es doblar la velocidad del flujo de datos. Esto se ha visto que se consigue colocando en el array coeficientes estáticos. En este caso es necesario añadir un registro a cada par de células para mantener el dato, esto indica que cada par de células del array serán idénticas. La cadena de acumulación para este array no tiene por lo tanto células idénticas, sino pares de células idénticas.

4.1.3 Aplicaciones.

Las operaciones de multiplicación de vectores y matrices son importantes en los niveles bajos e intermedios del procesamiento de la señal. Existen por lo tanto diversas aplicaciones en las que las arquitecturas sistólicas estudiadas pueden ser empleadas.

Convolución

Para la realización de un "convolver", podemos emplear un array de producto interno con ligeras modificaciones. Aquí una salida dada ***Yn*** es dada por el producto interno de un vector de coeficientes por un vector de los valores de entrada retrasados.

$$Y_n = \sum_{m=1}^{N} A_m X_i$$

en donde $i = n - m$.

En la Fig. 4.16 se muestra la estructura necesaria para realizar la convolución

[WhCa82][UrWo84]. Los datos son alimentados en serie desde la derecha en la fila inferior del array principal. Como los datos se desplazan en cada fila desde la derecha a la izquierda, cuando el primer bit introducido desde la derecha aparece en el lado izquierdo del array, es conducido al comienzo de la fila inmediatamente superior. De esta forma si una palabra X_i es introducida en la fila inferior del array principal interactuará con un paralelogramo de productos parciales que tienen acumulado Y_n. Si X_i es introducido, entonces contribuirá al paralelogramo de productos parciales con el término Y_{n+1}.

Para que el array funcione correctamente es evidente la necesidad de introducir un número (j) correcto de retrasos entre filas. Este número de retraso entre filas adyacentes n va a depender del alto y del ancho del array de productos internos. El valor del número de retrasos j va a depender del tipo de acumulador utilizado. En el caso de que el valor de j sea menor que cero, es evidente que el array no podrá operar.

$$\textit{Acumuladordoble} \quad j = B + \log_2 N - C - 1$$
$$\textit{Acumuladorsimple} \quad j = B + \log_2 N - 1$$

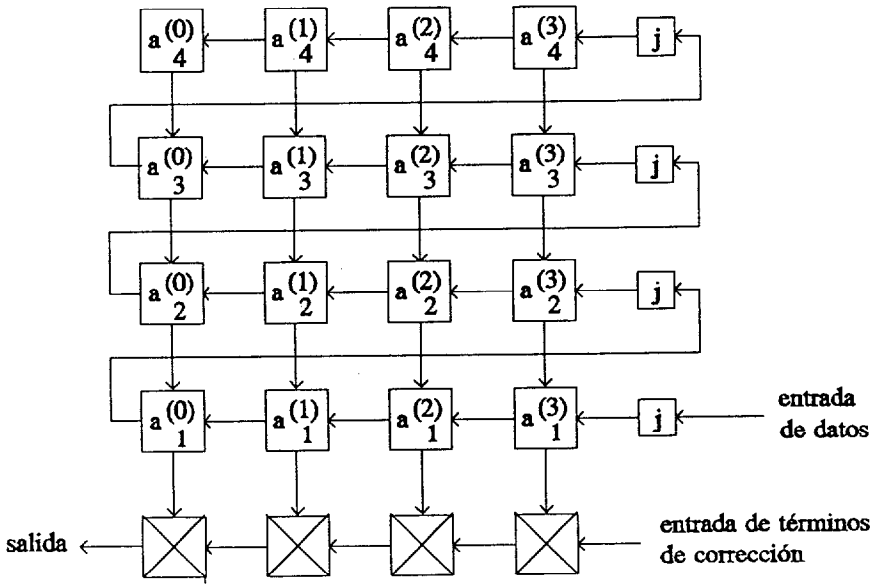

Fig. 4.16 Convolver.

Filtro IIR

La idea básica utilizada en la realización de un "convolver" puede ser utilizada para realizar un filtro IIR. En el caso de un filtro IIR, la salida Y_n es dada por el producto interno de un vector de coeficientes y un vector consistente de las entradas retrasadas y de las propias salidas.

$$Y_n = \sum_{m=1}^{N} A_m X_i + \sum_{m+1}^{M} b_m Y_i$$

en donde $i = n - m$.

Podemos considerar el filtro IIR como un simple array de productos internos en el cual multiplicamos un vector de coeficientes por un vector formado por las entradas retrasadas y por las propias salidas del filtro, de esta forma, las entradas y salidas se mueven de forma equivalente al "convolver", tal como podemos ver en la Fig. 4.17.

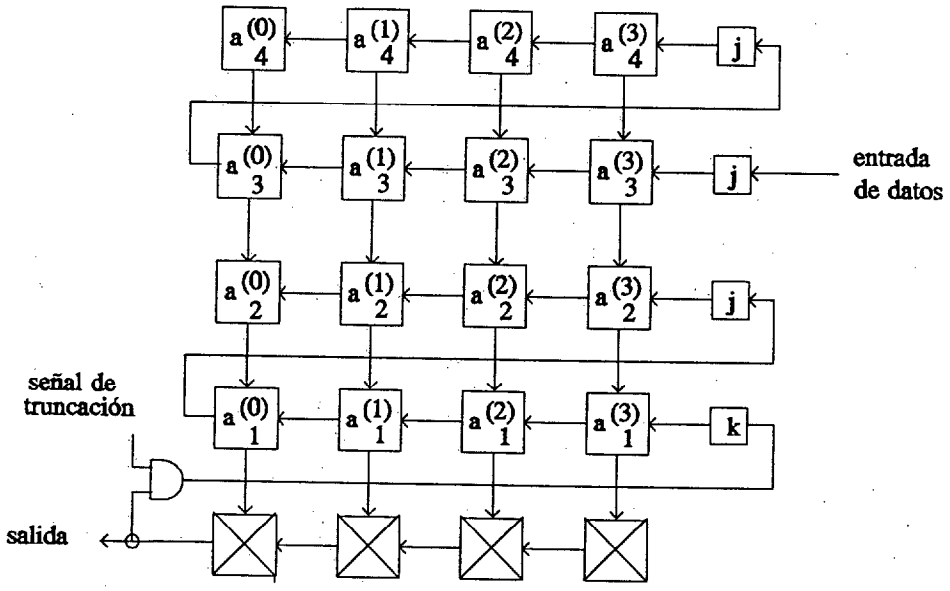

Fig. 4.17 Array de Filtro IIR.

Hay que tener en consideración que tenemos que recortar las salidas del filtro para poder realimentar éstas junto con las entradas, ya que el ancho de palabra de las salidas siempre es mayor que el de las entradas. El truncamiento se realiza dejando los B bits más significativos.

Ya que las salidas son realimentadas de nuevo en el array, sólo podemos considerar acumuladores simples. La longitud de la banda de ceros de guarda que se han de aplicar a la entrada será por lo tanto de valor $log_2 (N + M) + C$.

Además de los j retrasos que se requieren entre las filas del filtro, se requiere otro retraso k entre las salidas del acumulador y las Y entradas del array del filtro.

4.2 Ejemplo de filtro FIR sistólico.

En este apartado vamos a realizar una aplicación de la metodología propuesta tomando como ejemplo de referencia el diseño de filtros FIR sistólicos que trabajan a nivel de bit. El tipo de filtros que se van a desarrollar se realizarán utilizando como núcleo de la estructura un array multiplicador de productos internos que nos permite realizar las operaciones de sumas de productos, tal como se explicó en el capítulo 3. En esta implementación utilizaremos coeficientes estáticos y el array trabajará en complemento 2. La estructura final de estos filtros está pensada para su fácil realización en un circuito integrado cuya lógica de control externa sea mínima.

La presentación del diseño se realizará en tres partes:

1) Presentación de una implementación del filtro propuesto realizada con las herramientas Cadence, utilizando la librería de células estándar de 1.5 µm de ES2.

2) Descripción LDAP del filtro.

3) Programa de generación automática de filtros. Para la creación de este programa se ha utilizado la descripción LDAP compilada del filtro.

4.2.1 Descripción del mecanismo básico de trabajo.

En el apartado 4.1 se ha realizado un estudio general de los métodos de multiplicación de vectores y matrices utilizando arrays sistólicos. Este estudio se particularizó para arrays que operan a nivel de bit en el apartado 4.1.2, y se estudio varias aplicaciones en el apartado 4.1.3. El mecanismo básico de trabajo del filtro que se ha diseñado, se encuentra explicado ampliamente en el apartado 4.1.2, por lo que aquí solo comentaremos los detalles de la implementación realizada con la librería ES2 de 1.5µm. El diagrama general de bloques del circuito se encuentra representado en Fig. 4.18.

El filtro que se ha diseñado trabaja con 16 coeficientes de 16 bit y muestras de 12 bits. El bloque principal de este filtro está compuesto de dos arrays que se encargarán uno del cálculo de los productos parciales y el otro de su suma (**inner_product_array**). El array superior, de 16 filas por 16 columnas, se encarga de almacenar los bits de los coeficientes W de forma estática y efectuar el producto con los vectores de muestras X que le llegan por el Este. El array inferior, de 1 fila por 16 columna, recoge estos resultados y efectúa la suma.

Tal como se ha estudiado en el apartado 4.1.2, para trabajar en complemento 2 necesitamos introducir un array de señales de control que indiquen en cada célula si tiene que realizar en ese momento la inversión del signo de la operación. Además tenemos que introducir a través de la primera célula del acumulador una cadena de bits de corrección que se sumen al resultado obtenido en el array para obtener el dato final. Este término de corrección se introducirá a través de la señal **s_inp**.

La célula básica que compone el array de productos parciales se representa en la Fig. 4.19

y la célula que forma el array acumulador se representa en la Fig. 4.20.

Además de los dos arrays comentados para construir un filtro FIR, necesitamos recircular los bits de las muestras tal como se explica en el ejemplo de realización de un convolver del apartado 4.1.3. Para lograr la recirculación de los bit de las muestras es necesario construir un registro de desplazamiento (**delays_array**) con un número de retardos igual a **j**. El bus de control **ctrl<1:0>** permite variar el número de retrasos asociados al array de registros.

B = 12 (Bits Muestras X)
C = 16 (Bits Coeficientes W)
N = 16 (Numero de filas del array)
Retraso de los registros => $j = B + \log_2 N - 1 = 15$
Ceros de guarda => $z = \log_2 N + C = 20$

Como las muestras entrarán a través de un bus externo de 12 bit (**x_inp<11:0>**), es necesario añadir al sistema un circuito de carga paralelo-serie que se ha denominado **input_register**. Este bloque es el responsable de la introducción de los **z** ceros de guarda, siendo la señal **ctrl_data** la que controla este proceso.

Para sacar al exterior el resultado del filtro, nos hace falta disponer de un circuito serie-paralelo (**output_register**) que nos permita sacar los 32 bit del resultado del filtro a través de un bus de 16 líneas de salida (**x_out<15:0>**). La señal **ctrl_out** controla el proceso de multiplexación de la palabra de salida.

La carga de los coeficientes del filtro y de las señales de control que permiten las operaciones en complemento 2, se introducen en el circuito a través del bus **w_ctrl<15:0>**. La carga de coeficientes se realiza a través de la señal **load**.

La señal **x_out** se utiliza para el test del array principal. En esta señal observaremos los bits de las muestras una vez que han atravesado totalmente las 16 filas del array de productos parciales.

Una vez verificado el correcto funcionamiento del circuito, se realizó el posicionamiento y conexionado automático. El layout final del circuito se encuentra representado en la Fig. 4.25.

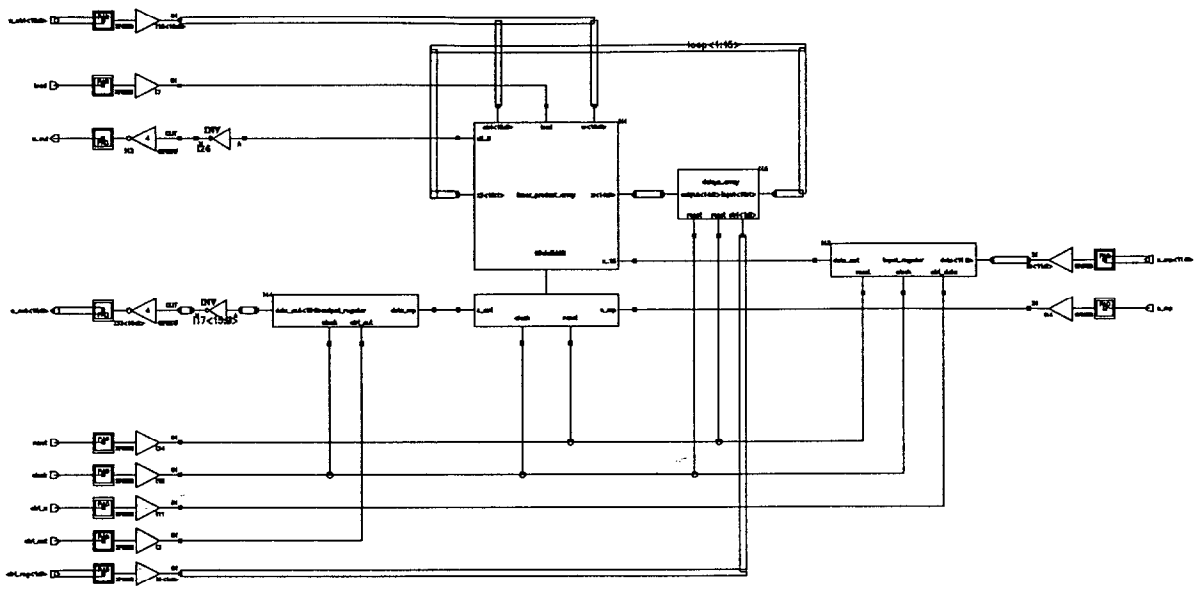

Fig. 4.18 Diagrama de bloques general del circuito.

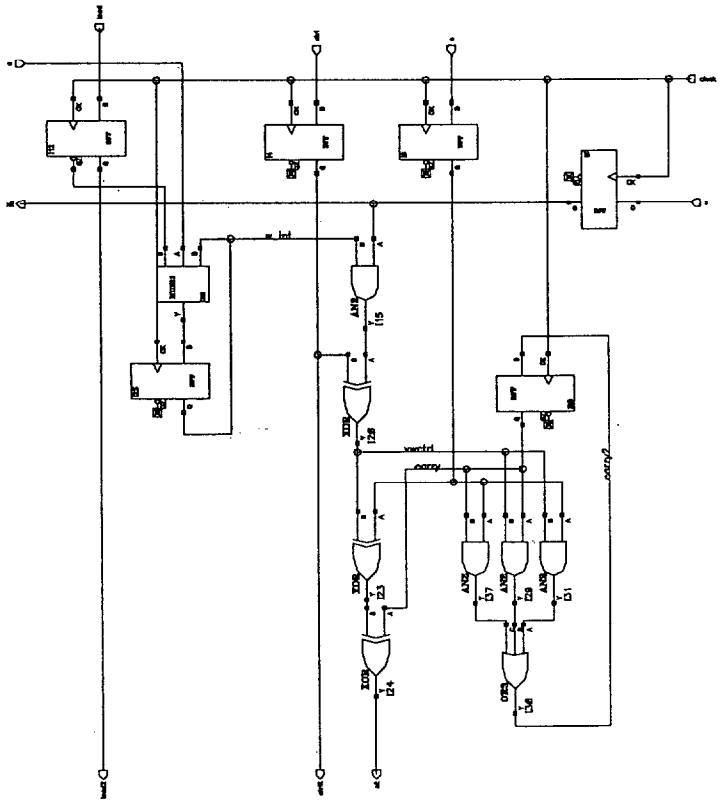

Fig. 4.19 Célula principal del array de productos parciales.

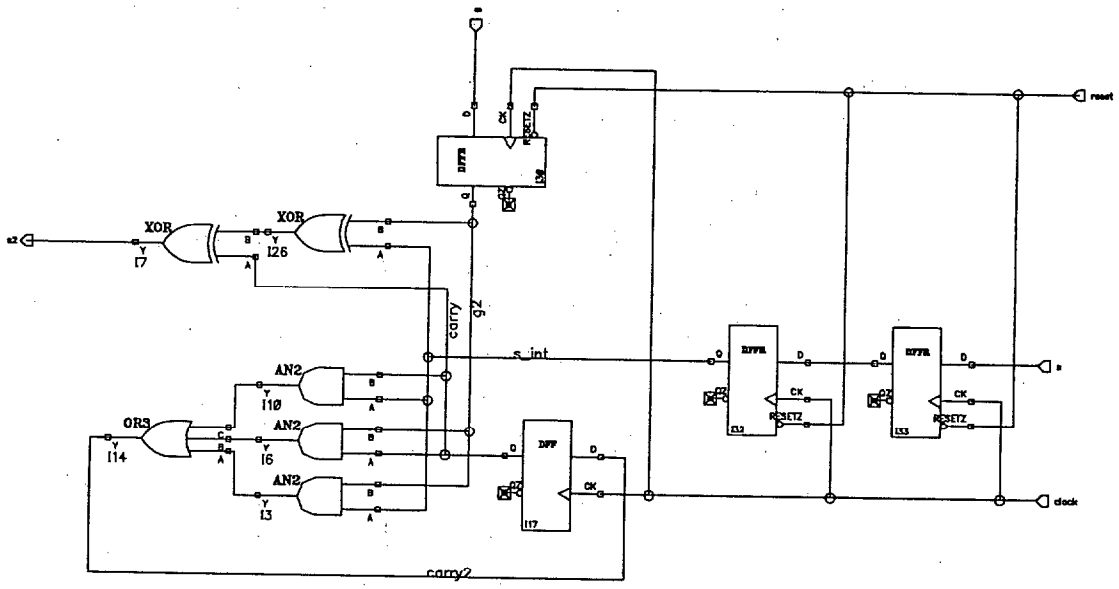

Fig. 4.20 Célula principal del array acumulador.

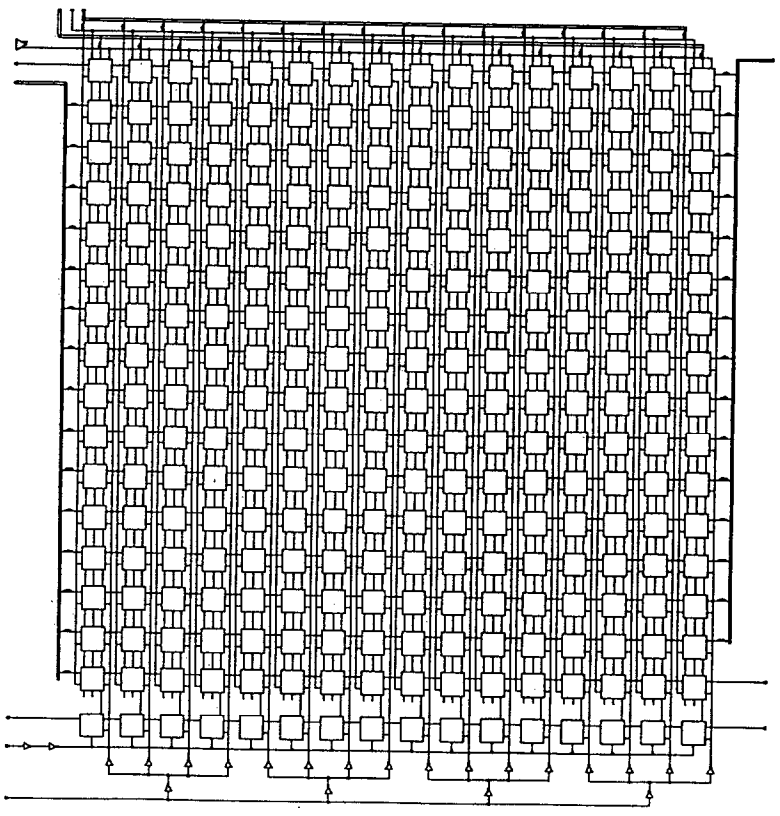

Fig. 4.21 Array de productos parciales y array acumulador.

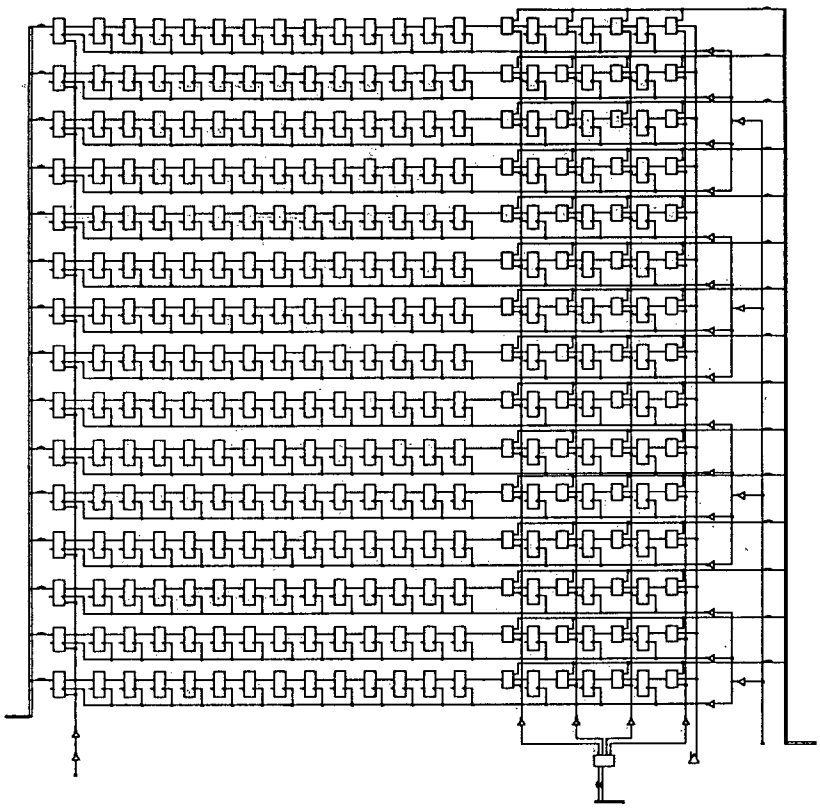

Fig. 4.22 Banco de registros de desplazamiento.

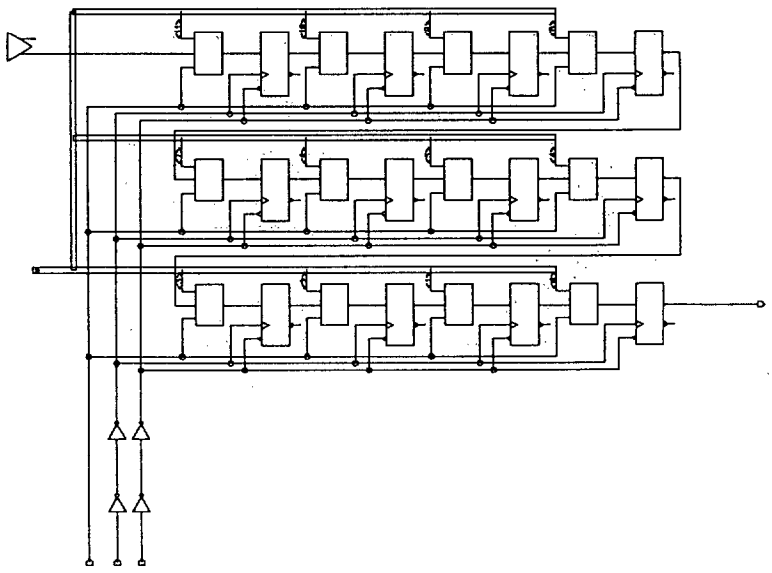

Fig. 4.23 Registro de entrada paralelo-serie.

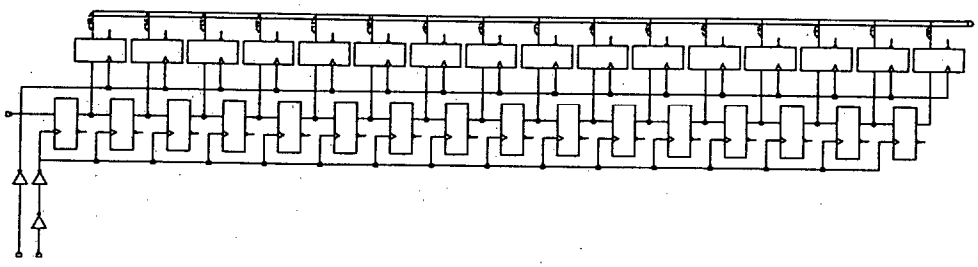

Fig. 4.24 Registro de salida serie-paralelo.

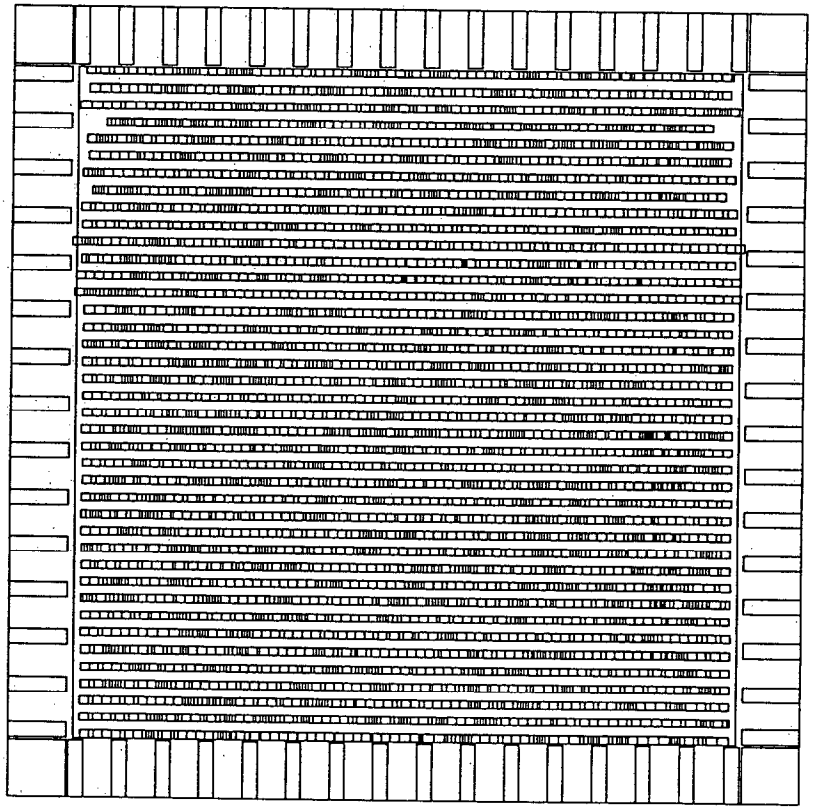

Fig. 4.25 Layout del circuito.

A continuación tenemos uno de los ficheros de simulación en los que se realiza la inicialización del circuito, carga de coeficientes y simulación del filtro trabajando en aritmética binaria positiva.

```
;
; Fichero de estimulos INPUT.STL
;
stlinit
settarget silos
;
; Entradas
;
defpin reset        in
defpin load         in
defpin ctrl_x       in
defpin ctrl_reg<1:0> in
defpin ctrl_out     in
defpin s_inp        in
defpin x_inp<11:0>  in
defpin w_ctrl<15:0> in
defpin clock        clk
defpin x_out        out
defpin s_out<15:0>  out
;
defformat reset load ctrl_x ctrl_reg:hex ctrl_out s_inp x_inp:hex w_ctrl:hex x_out
s_out:hex
deftiming 0.1ns 2.5ns 50ns
defclock "..........1111111111" clock
;
; Secuencia de desplazamiento de datos
;
defseq ciclo_trabajo
;
; B = 12 (Bits Muestras X)
; C = 16 (Bits Coeficientes W)
; N = 16 (Numero de filas del array)
; Retraso de los registros => j = B + log2 N - 1 = 15
; Ceros de guarda          => z = log2 N + C = 20
;
v 1 0 0 0x1 1 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 1 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
v 1 0 0 0x1 0 0 0x000 0x0000 ? ?
endseq ciclo_trabajo
;
; Comienzan los vectores
;
deftest
;
; Inicializamos el sistema.
;
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
;
; carga de coeficientes w en la matriz
; a la vez continuamos reseteando el resto de la logica.
;
xv 0 1 0 0x1 1 0 0x000 0xffff ? ?
xv 0 0 0 0x1 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x1 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x1 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x1 1 0 0x000 0x5689 ? ?
xv 0 0 0 0x1 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x1 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x1 1 0 0x000 0x56ff ? ?
xv 0 0 0 0x1 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x1 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x1 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x1 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x1 1 0 0x000 0x1223 ? ?
xv 0 0 0 0x1 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x1 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x1 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x1 1 0 0x000 0x00ff ? ?
;
; continuamos inicializando el resto del array
;
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
xv 0 0 0 0x3 1 0 0x000 0x0000 ? ?
;
; MUESTRA 1
xv 1 0 1 0x1 0 0 0x0ff 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 2
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
```

```
; MUESTRA 3
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 4
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 5
xv 1 0 1 0x1 0 0 0xfff 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 6
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 7
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 8
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 9
xv 1 0 1 0x1 0 0 0x001 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 10
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 11
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 12
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 13
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 14
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 15
xv 1 0 1 0x1 0 0 0xfff 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 16
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 17
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 18
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 19
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 20
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 21
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 22
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 23
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 24
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 25
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 26
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 27
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 28
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 29
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 30
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 31
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 32
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 33
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 34
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 35
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 36
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 37
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 38
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 39
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 40
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 41
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo
; MUESTRA 42
xv 1 0 1 0x1 0 0 0x000 0x0000 ? ?
callseq ciclo_trabajo

endtest
```

4.2.2 Descripción LDAP del filtro.

El fichero que se presenta a continuación es la descripción LDAP de la arquitectura del filtro sistólico realizada en el apartado anterior.

```
/* Descripcion LDAP de un filtro FIR sistolico */

USE default_level;
SIMUL  VHDL = 'fir.vhdl';

# Celula principal del array que forma el producto interno.
CELL Cell_inner {
   INOUT {
      load:IN > load2:OS;
      ctrl:IN > ctrl2:OS;
      s:IN > s2:OS;
      w:GW;
      x:IE > x2:OW;
      clock:GE;
   }
   FUNCTION VHDL='Cell_inner.vhdl';
   MODEL {
      V_VHDL='Descripcion de la celula Cell_inner':'Cell_inner.vhdl';
   }
}

# Celula basica del array acumulador.
CELL Cell_acu {
   INOUT {
      g:IN;
      s:IE > s2:OW;
      clock:GE;
      reset:GE;
   }
   FUNCTION VHDL = 'Cell_acu.vhdl';
   MODEL {
      V_VHDL='Descripcion de la celula Cell_acu ' : 'Cell_acu.vhdl';
   }
}

# Celula basica del array de registros de desplazamiento.
CELL Cell_reg {
   INOUT {
      in:IE > out:OW;
      clock:GE;
      reset:GE;
   }
   FUNCTION VHDL = 'Cell_reg.vhdl';
   MODEL {
      V_VHDL = 'Descripcion de la celula: Cell_reg' : 'Cell_reg.vhdl';
   }
}

# Celula de carga de datos de entrada
CELL input_register {
   INOUT { in:IE > out:OW; reset:IS; clock:IS; ctrl_data:IS }
   FUNCTION VHDL = 'input_register.vhdl';
   MODEL { V_VHDL = 'Descripcion de la celula de carga de dato' :
'input_register.vhdl' }
}
# Celula de datos de salida
CELL output_register {
   INOUT { data_inp:IE > data_out:OW; clock:IS; ctrl_out:IS }
   FUNCTION VHDL = 'output_register.vhdl';
   MODEL { V_VHDL='Descripcion de la celula de salida del
dato':'output_register.vhdl' }
}
# Definiendo pin de entrada.
CELL pin_inp {
   INOUT { pin:IW }
   MODEL { V_PINEXT = 'PIN INPUT' }
}

# Definiendo pin de salida.
CELL pin_out {
   INOUT { pin:OE }
   MODEL { V_PINEXT = 'PIN OUTPUT' }
}

# Generacion de los simbolos de todas las celulas
AUTO { Cell_inner; Cell_acu; Cell_reg; input_register; output_register; pin_inp;
pin_out }


/* Array de formacion de los productos internos */
ARRAY Array_inner { R[0:15][0:15] = Cell_inner }

/* Array de celulas acumuladoras */
ARRAY Array_acu { L[0][0:15] = Cell_acu }

/* Array de registros de desplazamiento */
ARRAY Array_reg { R[0:15][0:14] = Cell_reg }


SHEET s1 {
   PLACE {
      /* Arrays */
      acu   = Array_acu:[1000,1000]:[3000,500]:0;
      inner = Array_inner:[1000,2000]:[3000,3000]:0;
      reg   = Array_reg:[4500,2000]:[1500,3000]:0;

      /* Celulas */
      inp_reg = input_register:[6500,2000]:[500,500]:0;
      out_reg = output_register:[100,1000]:[500,500]:0;

      /* Pines de Entrada y Salida */
      ctrl_reg = pin_inp:[100,2000]:[100,100]:0;
      ctrl_out = pin_inp:[100,2300]:[100,100]:0;
      ctrl_x   = pin_inp:[100,2600]:[100,100]:0;
      clock    = pin_inp:[100,2900]:[100,100]:0;
      reset = pin_inp:[100,3200]:[100,100]:0;
      s_out    = pin_out:[100,3500]:[100,100]:0;
      x_out    = pin_out:[100,3800]:[100,100]:0;
      load  = pin_inp:[100,4100]:[100,100]:0;
      w_ctrl   = pin_inp:[100,4400]:[100,100]:0;
      x_inp    = pin_inp:[7000,4700]:[100,100]:45;
      s_inp    = pin_inp:[7000,5000]:[100,100]:45;
   }
   CONNECT {
      /* Conexiones entre arrays */
      inner[7][0:15].s2  = acu[0][0:15].g;
      inner[0:15][15].x  = reg[0:15][0].out;
      inner[0:14][0].x2  = reg[1:14][14].in;

      /* Relojes */
      inner[0:15][0*7].clock = clock.pin;
      reg[0:15][0*14].clock = clock.pin;
      acu[0][0:15].clock = clock.pin;
      inp_reg.clock = clock.pin;
      out_reg.clock = clock.pin = clock;

      /* Reset */
      inp_reg.reset = reset.pin = reset;
      reg[0:15][0*14].reset = reset.pin;
      acu[0][0:15].reset = reset.pin;

      /* Control */
      inner[0][0:15].ctrl = w_ctrl[0:15];
      inner[0][0:15].w = w_ctrl[0:15];
   }

}
```

4.2.3 Descripción LDV del filtro.

En este apartado presentamos el listado del fichero fir.ldv generado de forma automática por GeneSis. Esta descripción es leida y representada gráficamente por el programa SiMon.

```
# ************************************************
#
#     Fichero: 'filtro.ldv'
#     Realizado: 15/06/1992 [ Monday ] a las 10:19:43 AM
#
#     Generado por utilidad GENLDV...
#
# ************************************************

* default_level

CELL Cell_inner  191 191

   MODEL VHDL Cell_inner.vhdl


   PIN load     BIT    IN  6 54 167
   PIN load2    BIT    OS  6 54 24
   PIN ctrl     BIT    IN  6 95 167
   PIN ctrl2    BIT    OS  6 95 24
   PIN s        BIT    IN  6 136 167
   PIN s2       BIT    OS  6 136 24
   PIN w        BIT    GW  6 24 54
   PIN x        BIT    IE  6 167 95
   PIN x2       BIT    OW  6 24 95
   PIN clock    BIT    GE  6 167 136

   BOX  1 2 24 24 167 167

   LIN  3 54 155 64 167
   LIN  3 54 155 44 167
   LIN  3 54 167 54 191
   LABEL  10 1 52 165 load

   LIN  3 54 12 44 24
   LIN  3 54 12 64 24
   LIN  3 54 12 54 0
   LABEL  10 1 52 25 load2

   LIN  3 95 155 105 167
   LIN  3 95 155 85 167
   LIN  3 95 167 95 191
   LABEL  10 1 93 165 ctrl

   LIN  3 95 12 85 24
   LIN  3 95 12 105 24
   LIN  3 95 12 95 0
   LABEL  10 1 93 25 ctrl2

   LIN  3 136 155 146 167
   LIN  3 136 155 126 167
   LIN  3 136 167 136 191
   LABEL  10 1 136 165 s

   LIN  3 136 12 126 24
   LIN  3 136 12 146 24
   LIN  3 136 12 136 0
   LABEL  10 1 135 25 s2

   LIN  3 24 54 0 54
   LIN  3 36 64 36 44
   LIN  3 24 64 36 64
   LIN  3 24 44 36 44
   LABEL  10 1 22 54 w

   LIN  3 155 95 167 85
   LIN  3 155 95 167 105
   LIN  3 167 95 191 95
   LABEL  10 1 168 95 x

   LIN  3 12 95 24 105
   LIN  3 12 95 24 85
   LIN  3 12 95 0 95
   LABEL  10 1 21 95 x2

   LIN  3 167 136 191 136
   LIN  3 155 126 155 146
   LIN  3 167 126 155 126
   LIN  3 167 146 155 146
   LABEL  10 1 168 136 clock

   TEXT  6 1 71 71 "Cell_inner"

END

CELL Cell_acu  191 191

   MODEL VHDL Cell_acu.vhdl


   PIN g        BIT    IN  6 95 167
   PIN s        BIT    IE  6 167 54
   PIN s2       BIT    OW  6 24 54
   PIN clock    BIT    GE  6 167 95
   PIN reset    BIT    GE  6 167 136

   BOX  1 2 24 24 167 167

   LIN  3 95 155 105 167
   LIN  3 95 155 85 167
   LIN  3 95 167 95 191
   LABEL  10 1 95 165 g

   LIN  3 155 54 167 44
   LIN  3 155 54 167 64
   LIN  3 167 54 191 54
   LABEL  10 1 168 54 s

   LIN  3 12 54 24 64
   LIN  3 12 54 24 44
   LIN  3 12 54 0 54
   LABEL  10 1 21 54 s2

   LIN  3 167 95 191 95
   LIN  3 155 85 155 105
   LIN  3 167 85 155 85
   LIN  3 167 105 155 105
   LABEL  10 1 168 95 clock

   LIN  3 167 136 191 136
   LIN  3 155 126 155 146
   LIN  3 167 126 155 126
   LIN  3 167 146 155 146
   LABEL  10 1 168 136 reset

   TEXT  6 1 71 71 "Cell_acu"

END

CELL Cell_reg  191 191

   MODEL VHDL Cell_reg.vhdl


   PIN in       BIT    IE  6 167 54
   PIN out      BIT    OW  6 24 54
   PIN clock    BIT    GE  6 167 95
   PIN reset    BIT    GE  6 167 136

   BOX  1 2 24 24 167 167

   LIN  3 155 54 167 44
```

```
  LIN 3 155 54 167 64
  LIN 3 167 54 191 54
  LABEL 10 1 168 54 in

  LIN 3 12 54 24 64
  LIN 3 12 54 24 44
  LIN 3 12 54 0 54
  LABEL 10 1 20 54 out

  LIN 3 167 95 191 95
  LIN 3 155 85 155 105
  LIN 3 167 85 155 85
  LIN 3 167 105 155 105
  LABEL 10 1 168 95 clock

  LIN 3 167 136 191 136
  LIN 3 155 126 155 146
  LIN 3 167 126 155 126
  LIN 3 167 146 155 146
  LABEL 10 1 168 136 reset

  TEXT 6 1 71 71 "Cell_reg"

END

CELL input_register 191 191

  MODEL VHDL input_register.vhdl


  PIN in       BIT   IE 6 167 95
  PIN out      BIT   OW 6 24 95
  PIN reset    BIT   IS 6 54 24
  PIN clock    BIT   IS 6 95 24
  PIN ctrl_data BIT  IS 6 136 24

  BOX 1 2 24 24 167 167

  LIN 3 155 95 167 85
  LIN 3 155 95 167 105
  LIN 3 167 95 191 95
  LABEL 10 1 168 95 in

  LIN 3 12 95 24 105
  LIN 3 12 95 24 85
  LIN 3 12 95 0 95
  LABEL 10 1 20 95 out

  LIN 3 54 36 44 24
  LIN 3 54 36 64 24
  LIN 3 54 24 54 0
  LABEL 10 1 52 25 reset

  LIN 3 95 36 85 24
  LIN 3 95 36 105 24
  LIN 3 95 24 95 0
  LABEL 10 1 93 25 clock

  LIN 3 136 36 126 24
  LIN 3 136 36 146 24
  LIN 3 136 24 136 0
  LABEL 10 1 132 25 ctrl_data

  TEXT 6 1 71 71 "input_register"

END

CELL output_register 150 150

  MODEL VHDL output_register.vhdl


  PIN data_inp  BIT   IE 6 126 75
  PIN data_out  BIT   OW 6 24 75
  PIN clock     BIT   IS 6 54 24
  PIN ctrl_out  BIT   IS 6 95 24

  BOX 1 2 24 24 126 126

  LIN 3 114 75 126 65
  LIN 3 114 75 126 85
  LIN 3 126 75 150 75
  LABEL 10 1 127 75 data_inp

  LIN 3 12 75 24 85
  LIN 3 12 75 24 65
  LIN 3 12 75 0 75
  LABEL 10 1 15 75 data_out

  LIN 3 54 36 44 24
  LIN 3 54 36 64 24
  LIN 3 54 24 54 0
  LABEL 10 1 52 25 clock

  LIN 3 95 36 85 24
  LIN 3 95 36 105 24
  LIN 3 95 24 95 0
  LABEL 10 1 91 25 ctrl_out

  TEXT 6 1 51 51 "output_register"

END

CELL pin_inp 109 109

  MODEL PINEXT


  PIN pin      BIT   IW 6 24 54

  BOX 1 2 24 24 85 85

  LIN 3 36 54 24 64
  LIN 3 36 54 24 44
  LIN 3 24 54 0 54
  LABEL 10 1 20 54 pin

  TEXT 6 1 30 30 "pin_inp"

END

CELL pin_out 109 109

  MODEL PINEXT


  PIN pin      BIT   OE 6 85 54

  BOX 1 2 24 24 85 85

  LIN 3 97 54 85 44
  LIN 3 97 54 85 64
  LIN 3 97 54 109 54
  LABEL 10 1 86 54 pin

  TEXT 6 1 30 30 "pin_out"

END

BLOCK Array_inner

  PLACE cell(0,0)     0 15000  0 Cell_inner    1000 1000
  PLACE cell(0,1)     1000 15000  0 Cell_inner    1000 1000
  ...
  ...
  ...
  ...
  PLACE cell(15,14)   14000 0  0 Cell_inner    1000 1000
  PLACE cell(15,15)   15000 0  0 Cell_inner    1000 1000

END

BLOCK Array_acu

  PLACE cell(0,0)     0 0  0 Cell_acu      1000 1000
  PLACE cell(0,1)     1000 0  0 Cell_acu      1000 1000
  PLACE cell(0,2)     2000 0  0 Cell_acu      1000 1000
  PLACE cell(0,3)     3000 0  0 Cell_acu      1000 1000
  PLACE cell(0,4)     4000 0  0 Cell_acu      1000 1000
  PLACE cell(0,5)     5000 0  0 Cell_acu      1000 1000
  PLACE cell(0,6)     6000 0  0 Cell_acu      1000 1000
  PLACE cell(0,7)     7000 0  0 Cell_acu      1000 1000
  PLACE cell(0,8)     8000 0  0 Cell_acu      1000 1000
  PLACE cell(0,9)     9000 0  0 Cell_acu      1000 1000
  PLACE cell(0,10)    10000 0  0 Cell_acu     1000 1000
  PLACE cell(0,11)    11000 0  0 Cell_acu     1000 1000
  PLACE cell(0,12)    12000 0  0 Cell_acu     1000 1000
  PLACE cell(0,13)    13000 0  0 Cell_acu     1000 1000
  PLACE cell(0,14)    14000 0  0 Cell_acu     1000 1000
  PLACE cell(0,15)    15000 0  0 Cell_acu     1000 1000

END

BLOCK Array_reg

  PLACE cell(0,0)     0 15000  0 Cell_reg     1000 1000
  PLACE cell(0,1)     1000 15000  0 Cell_reg     1000 1000
```

```
  ...
  ...
  ...
  ...
  PLACE cell(15,13)   13000 0  0 Cell_reg      1000 1000
  PLACE cell(15,14)   14000 0  0 Cell_reg      1000 1000

END

SHEET s1

  PLACE acu          1000 1000  0 Array_acu      3000 500
  PLACE inner        1000 2000  0 Array_inner    3000 3000
  PLACE reg          4500 2000  0 Array_reg      1500 3000
  PLACE inp_reg      6500 2000  0 input_register  500 500
  PLACE out_reg      100 1000   0 output_register 500 500
  PLACE ctrl_reg     100 2000   0 pin_inp        100 100
  PLACE ctrl_out     100 2300   0 pin_inp        100 100
  PLACE ctrl_x       100 2600   0 pin_inp        100 100
  PLACE clock        100 2900   0 pin_inp        100 100
  PLACE reset        100 3200   0 pin_inp        100 100
  PLACE s_out        100 3500   0 pin_out        100 100
  PLACE x_out        100 3800   0 pin_out        100 100
  PLACE load         100 4100   0 pin_inp        100 100
  PLACE w_ctrl       100 4400   0 pin_inp        100 100
  PLACE x_inp        7000 4700  45 pin_inp        100 100
  PLACE s_inp        7000 5000  45 pin_inp        100 100

  TEXT 9 1 0 0 "s1"

END

NETLIST
  NODE s1_0
    s1.inner.cell(0,0).x
    s1.inner.cell(0,1).x2
  NODE s1_1
    s1.inner.cell(0,1).x
    s1.inner.cell(0,2).x2
  NODE s1_2
    s1.inner.cell(0,2).x
    s1.inner.cell(0,3).x2
  NODE s1_3
    s1.inner.cell(0,3).x
    s1.inner.cell(0,4).x2
  NODE s1_4
    s1.inner.cell(0,4).x
    s1.inner.cell(0,5).x2
  NODE s1_5
    s1.inner.cell(0,5).x
    s1.inner.cell(0,6).x2
  NODE s1_6
    s1.inner.cell(0,6).x
    s1.inner.cell(0,7).x2
  NODE s1_7
    s1.inner.cell(0,7).x
    s1.inner.cell(0,8).x2
  NODE s1_8
    s1.inner.cell(0,8).x
    s1.inner.cell(0,9).x2
  NODE s1_9
    s1.inner.cell(0,9).x
    s1.inner.cell(0,10).x2
  NODE s1_10
    s1.inner.cell(0,10).x
    s1.inner.cell(0,11).x2
  NODE s1_11
    s1.inner.cell(0,11).x
    s1.inner.cell(0,12).x2

  ...
  ...
  ...
  ...
```

4.2.4 Programa generador de filtro.

A partir de la descripción LDAP del filtro, se ha generado la siguiente descripción en base a macrofunciones C.

```
/*
   ************************************************************

       Fichero: 'FILTRO.C'
      Realizado: 15/06/1992 [ Monday ] a las 10:19:38 AM

      Generado por interprete Lexer...

   ************************************************************
*/

#include <stdio.h>
#include "basico.h"     /* Definiciones propias del sistema */

/* ======================================== */
/* Funcion principal a llamar por el nucleo */
/* ======================================== */

Main_Sheet()
{

   /* === Inicializacion del sistema === */

   Init_System();

   Use( "default_level" );

   /* === Definicion de ficheros para simuladores === */

   Def_Simul(V_VHDL,"lir.vhdl");

   /* === Definicion de Cell_inner === */

   Init_Cell("Cell_inner");

      Def_Pin("load", IN, T_BIT, 1);
      Def_Pin("load2", OS, T_BIT, 1);
      Def_Con("load", "load2");

      Def_Pin("ctrl", IN, T_BIT, 1);
      Def_Pin("ctrl2", OS, T_BIT, 1);
      Def_Con("ctrl", "ctrl2");

      Def_Pin("s", IN, T_BIT, 1);
      Def_Pin("s2", OS, T_BIT, 1);
      Def_Con("s", "s2");

      Def_Pin("w", GW, T_BIT, 1);
      Def_Pin("x", IE, T_BIT, 1);
      Def_Pin("x2", OW, T_BIT, 1);
      Def_Con("x", "x2");

      Def_Pin("clock", GE, T_BIT, 1);
      Function(V_VHDL, "Cell_inner.vhdl");
      Def_Model( V_VHDL, "Descripcion de la celula Cell_inner", NULL,
"Cell_inner.vhdl");

   End_Cell();


   /* === Definicion de Cell_acu === */

   Init_Cell("Cell_acu");

      Def_Pin("g", IN, T_BIT, 1);
      Def_Pin("s", IE, T_BIT, 1);
      Def_Pin("s2", OW, T_BIT, 1);
      Def_Con("s", "s2");

      Def_Pin("clock", GE, T_BIT, 1);
      Def_Pin("reset", GE, T_BIT, 1);
      Function(V_VHDL, "Cell_acu.vhdl");
      Def_Model( V_VHDL, "Descripcion de la celula Cell_acu ", NULL,
"Cell_acu.vhdl");

   End_Cell();


   /* === Definicion de Cell_reg === */

   Init_Cell("Cell_reg");

      Def_Pin("in", IE, T_BIT, 1);
      Def_Pin("out", OW, T_BIT, 1);
      Def_Con("in", "out");

      Def_Pin("clock", GE, T_BIT, 1);
      Def_Pin("reset", GE, T_BIT, 1);
      Function(V_VHDL, "Cell_reg.vhdl");
      Def_Model( V_VHDL, "Descripcion de la celula: Cell_reg", NULL,
"Cell_reg.vhdl");

   End_Cell();


   /* === Definicion de input_register === */

   Init_Cell("input_register");

      Def_Pin("in", IE, T_BIT, 1);
      Def_Pin("out", OW, T_BIT, 1);
      Def_Con("in", "out");

      Def_Pin("reset", IS, T_BIT, 1);
      Def_Pin("clock", IS, T_BIT, 1);
      Def_Pin("ctrl_data", IS, T_BIT, 1);
      Function(V_VHDL, "input_register.vhdl");
      Def_Model( V_VHDL, "Descripcion de la celula de carga de dato", NULL,
"input_register.vhdl");

   End_Cell();


   /* === Definicion de output_register === */

   Init_Cell("output_register");

      Def_Pin("data_inp", IE, T_BIT, 1);
      Def_Pin("data_out", OW, T_BIT, 1);
      Def_Con("data_inp", "data_out");

      Def_Pin("clock", IS, T_BIT, 1);
      Def_Pin("ctrl_out", IS, T_BIT, 1);
      Function(V_VHDL, "output_register.vhdl");
      Def_Model( V_VHDL, "Descripcion de la celula de salida del dato", NULL,
"output_register.vhdl");

   End_Cell();


   /* === Definicion de pin_inp === */

   Init_Cell("pin_inp");

      Def_Pin("pin", IW, T_BIT, 1);
      Def_Model( V_PINEXT, "PIN INPUT", NULL, "");

   End_Cell();


   /* === Definicion de pin_out === */

   Init_Cell("pin_out");

      Def_Pin("pin", OE, T_BIT, 1);
      Def_Model( V_PINEXT, "PIN OUTPUT", NULL, "");

   End_Cell();


   AutoSymbol( "Cell_inner", "Cell_acu", "Cell_reg", "input_register",
"output_register", "pin_inp", "pin_out", NULL );

   /* === Definicion de Array_inner === */
```

```
  Init_Array("Array_inner");

    Def_Rec( 0, 0, 15, 15, "Cell_inner");
  End_Array();

  /* === Definicion de Array_acu === */

  Init_Array("Array_acu");

    Def_Rec( 0, 0, 0, 15, "Cell_acu");
  End_Array();

  /* === Definicion de Array_reg === */

  Init_Array("Array_reg");

    Def_Rec( 0, 0, 15, 14, "Cell_reg");
  End_Array();
  /* === Definicion de s1 === */

  Init_Sheet("s1");
    Place( 1000, 1000, 0, 3000, 500, "acu", "Array_acu");
    Place( 1000, 2000, 0, 3000, 3000, "inner", "Array_inner");
    Place( 4500, 2000, 0, 1500, 3000, "reg", "Array_reg");
    Place( 6500, 2000, 0, 500, 500, "inp_reg", "input_register");
    Place( 100, 1000, 0, 500, 500, "out_reg", "output_register");
    Place( 100, 2000, 0, 100, 100, "ctrl_reg", "pin_inp");
    Place( 100, 2300, 0, 100, 100, "ctrl_out", "pin_inp");
    Place( 100, 2600, 0, 100, 100, "ctrl_x", "pin_inp");
    Place( 100, 2900, 0, 100, 100, "clock", "pin_inp");
    Place( 100, 3200, 0, 100, 100, "reset", "pin_inp");
    Place( 100, 3500, 0, 100, 100, "s_out", "pin_out");
    Place( 100, 3800, 0, 100, 100, "x_out", "pin_out");
    Place( 100, 4100, 0, 100, 100, "load", "pin_inp");
    Place( 100, 4400, 0, 100, 100, "w_ctrl", "pin_inp");
    Place( 7000, 4700, 45, 100, 100, "x_inp", "pin_inp");
    Place( 7000, 5000, 45, 100, 100, "s_inp", "pin_inp");

  End_Sheet();

}
```

4.2.5 Visualización del filtro en SiMon.

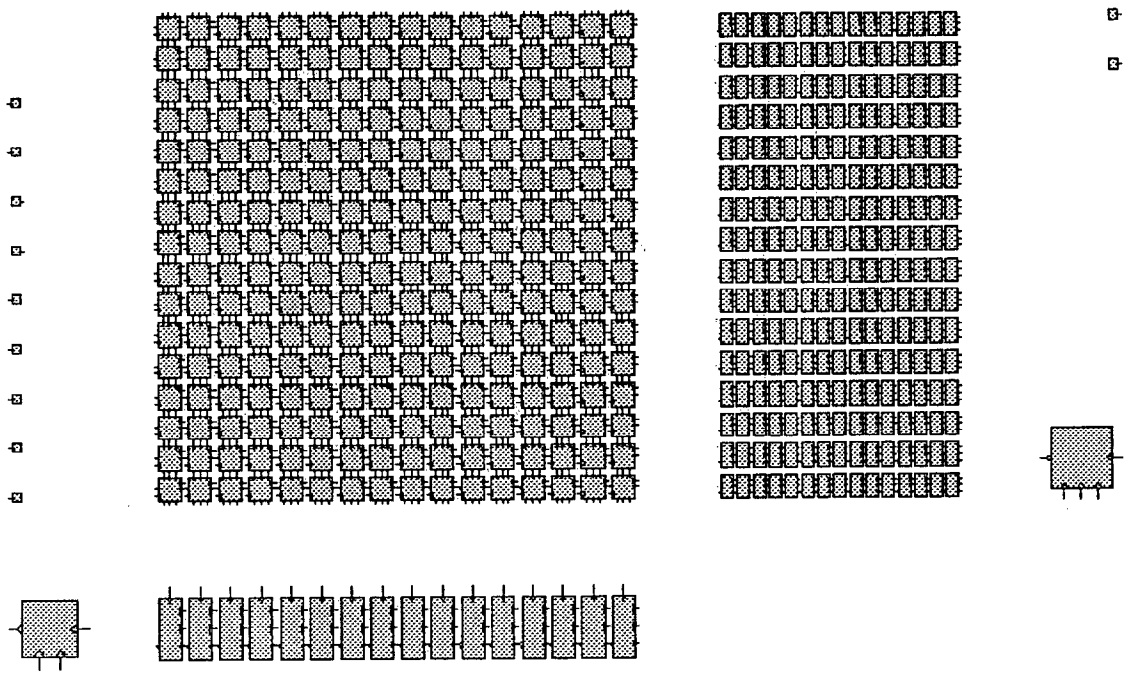

Fig. 4.26 Vista del filtro en una ventana de SiMon.

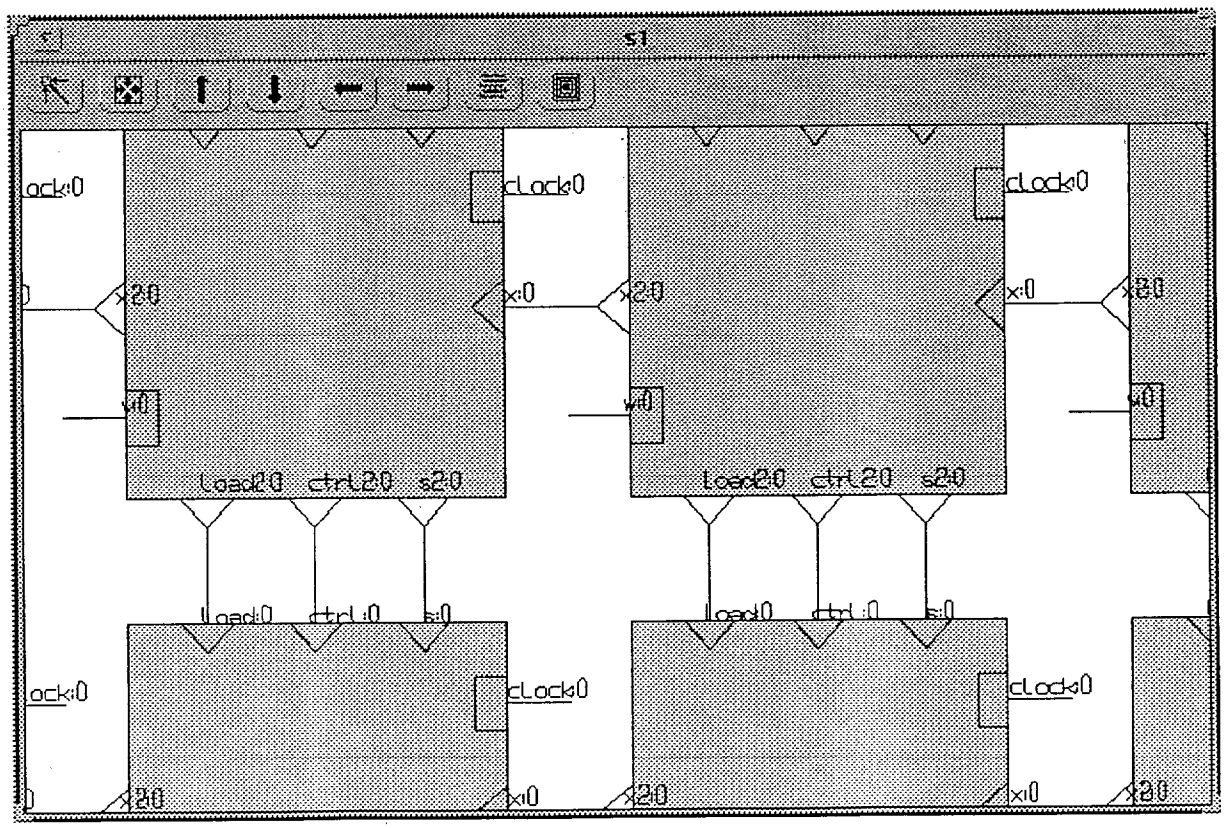

Fig. 4.27 Detalle de una sección del array de productos parciales.

Capítulo 5.

Implementación de los lenguajes y herramientas del entorno EASAP.

5.1 Lenguaje LDAP interpretado.

Tal como se estudió en el capítulo 3, la versión interpretada del lenguaje LDAP proporciona la posibilidad de poder realizar la descripción del sistema a través de un interfase de alto nivel. Esto se implementa con un intérprete de comandos denominado ***lexer***. La entrada para este intérprete viene dada por un fichero de descripción que recoge los datos del sistema. Por defecto, se tomará este fichero con extensión *'.src'*. Para acceder a las utilidades proporcionadas por el intérprete se puede trabajar directamente con él (programa ***lexer***), o bien, a través del programa ***Genesis***.

5.1.1 Sintaxis de los comandos.

Los comandos son las estructuras que nos permitirán definir los elementos básicos de nuestras arquitecturas. Son palabras clave que afectan a todo aquello que está englobado dentro de los caracteres delimitadores "{" y "}". Todo comando inicia su campo de aplicación con la llave "{". Entre un comando y su llave de inicio, podemos encontrar en algunos casos el nombre asignado a la estructura que define el comando. Un claro ejemplo de esto lo encontramos cuando definimos una célula. Aquí tendremos que utilizar el comando "CELL" seguido del nombre que se asocia a esa célula.

Línea de comando:

CELL nombre_cell { /* Definición de la célula */ };

Los distintos comandos que se describirán más adelante estarán en la mayoría de los casos incluidos dentro de otro comando.

Todos los comandos del lenguaje serán reconocidos únicamente en letras mayúsculas, pudiendo usarse nombres genéricos internos a la descripción y que coincidan con el de los comandos, siempre y cuando no estén escritos en mayúsculas. Así, por ejemplo, el comando **CELL** será reconocido como tal, y la palabra **Cell** no será confundida con éste.

Como norma general a seguir, los nombres de señales y buses serán escritos en minúsculas, los nombres de primitivas en mayúsculas y los comandos en mayúscula y negrita.

Caracteres separadores.

En general, se tomarán como caracteres separadores entre comandos y operadores todos

aquellos caracteres blancos, tabuladores y retornos de carro, siendo irrelevante el número de ellos que se incluyan.

Para marcar los finales de comando se usará, normalmente, el carácter ';'. Sin embargo, en determinadas ocasiones se podrá emplear otro carácter. Así, podrá usarse uno de los siguientes caracteres:

; Esta será la situación más normal e indicará simplemente el final de la definición del pin sin ninguna acción adicional a realizar.

/ Como se comentará posteriormente, en la descripción del sistema se permite el empleo de comentarios al estilo del C, señalados por /*...*/. Si este comentario se sitúa al final de la misma línea de la definición del pin, las normas sintáticas del lenguaje LDAP interpretado permiten que el ";" que normalmente marca el final de la definición se suprima, tomando como carácter delimitador la "/" del comienzo del comentario.

> Este carácter es un operador del LDAP interpretado, y su empleo indica que el pin cuya definición delimita se ha de autoconectar con el pin que se defina a continuación.

} Dado que varios comandos permiten en su interior diferentes definiciones, en la última de ellas se puede suprimir el ";" delimitador de final de definición de pin y usar como delimitador el carácter "}" que cierra el ámbito de aplicación del comando INOUT.

Inclusión de comentarios.

El interprete del LDAP admite dos tipos de comentarios. Por un lado, están aquellos comentarios que el usuario escribe en el fichero fuente como guía personal a la hora de leer y/o editar el contenido de este fichero. Funcionalmente, actúan igual que los comentarios que se ponen en el fuente de un lenguaje de programación (C, PASCAL, ...). Su contenido será simplemente ignorado, y su sintaxis es similar a la de los comentarios en C, es decir:

/ ... cualquier texto ... */*

Las normas que rigen la construcción de estos comentarios son las mismas que para el lenguaje C, es decir:

* No se permiten comentarios anidados.

* Dentro de un comentario puede haber cualquier combinación de caracteres a excepción de "/*" y "*/" (si aparecen ambos, estaríamos en el caso de comentario anidado).

* El contenido puede ocupar tantas líneas como se desee.

El segundo tipo de comentarios que se admiten en la descripción del sistema, también son ignorados por el analizador a efectos de análisis de su contenido pero, a diferencia del anterior tipo comentado, éste es enviado a la pantalla. Su utilidad reside en el hecho de permitir al usuario el mostrar, en la ventana de salida de texto, mensajes que le vayan indicando los puntos por los que va pasando el analizador léxico y, de esta forma, llevar un seguimiento de la evolución del proceso de generación del sistema a partir de la descripción dada por el

usuario. Al igual que en el caso anterior, hay una serie de normas que deben cumplir los comentarios que se construyan de este tipo:

* El comentario se marcará con un carácter "#" al comienzo de la linea.

* Ocuparán una sola línea de texto.

* En su contenido se admite cualquier combinación de caracteres, incluyendo el propio "#", siempre que no se exceda de la línea en que se comenzó el comentario.

Si se razona a partir de estas normas, se llega a la conclusión de que el comentario, para que sea reconocido como tal, debe de estar sólo en la línea en que se encuentra y que no puede haber ningún carácter (perteneciente a él o no) a la izquierda del carácter "#".

```
USE default_level;
SIMUL V_VHDL = 'output.vhdl';
...
CELL cell_1 {
        INOUT { ... }
        INTERNAL { ... }
        FUNCTION { ... }
        MODEL { ... }
}
...
CELL cell_n {
        INOUT { ... }
        FUNCTION { ... }
        DRAWS { ... }
}
ARRAY array_1 { ... }
...
ARRAY array_n { ... }

BLOCK block_1 {
        PLACE { ... }
        CONNECT { ... }
}
...
BLOCK block_n {
        DRAWS { ... }
}
SHEET sheet_1 {
        DRAWS { ... }
        PLACE { ... }
        CONNECT { ... }
}
...
SHEET sheet_2 {
        PLACE { ... }
        CONNECT { ... }
}
...
AUTO { cell_1; .... }
```

Fig. 5.1 Organización general de un fichero LDAP.

Tratamiento de buses.

Con respecto a cómo podemos definir las señales y los buses, éstos podrán tener un máximo de 32 caracteres. Sólo se permiten letras, números y underscore al escribir sus nombres. El número de hilos de un bus puede ser cualquiera, aunque se recomienda que no sea mayor de 32. La definición de un bus se realiza de la forma siguiente:

nombre_bus[entero:entero]

El primer número indica el índice de hilo mayor, y el segundo el índice menor. No se

produce ningún tipo de error al introducir blancos de separación entre el nombre del bus y los corchetes, o entre éstos y los valores enteros.

Estructura general de un fichero LDAP.

En Fig. 5.1 tenemos un ejemplo de la estructura general de un fichero LDAP. El orden de definición de los distintos elementos del lenguaje es indiferente. La única condición que se ha de cumplir es que se han de definir el primer lugar aquellos elementos que van a ser referenciados desde la definición de otro elemento posterior.

5.1.2 Comandos del lenguaje.

ARRAY:

Este comando permite la definición de un array de células. Vamos a poder definir tres tipos de arrays; array lineal, rectangular y hexagonal (Fig. 5.2). Las células octogonales a todos los efectos formarán arrays rectangulares en los cuales estarán permitidas las conexiones en diagonal. Hay que destacar que un array va a poder estar formado por células de distinto tipo, ya que es corriente que las células periféricas de un array sean diferentes de las internas. En la declaración de un array vamos a poder emplear distintos tipos de sentencias que hecen referencia a distintos tipos de células. La dimensión máxima del array que estamos definiendo vendrá dada por el conjunto de las sentencia de asignación de células. La sintaxis de este comando es:

```
ARRAY nombre_array {
        ...
        Tipo_array[a:b][c:d] = nombre_cell;
        ...
}
```

donde:

Tipo_Array Tipo de array a emplear. Los valores permitidos son:

L: Array Lineal
R: Array Rectangular.
H: Array Hexagonal.

[a:b][c:d] Rangos de variación en filas y columnas, respectivamente.

tipo_cel Tipo de la célula a situar en el array.

Ejemplo:

```
CELL Cell_1 {
        INOUT { w:IN > z:OS }
}
CELL Cell_2 {
        INOUT { a:IN > b:OS }
}
ARRAY Array_1 {
        R[1:3][0:3] = Cell_1;
```

```
        R[0][0:3] = Cell_2;
}
SHEET Sheet_1 {
        PLACE {
                array = Array_1:[0,0]:[100,100]:0;
        }
}
```

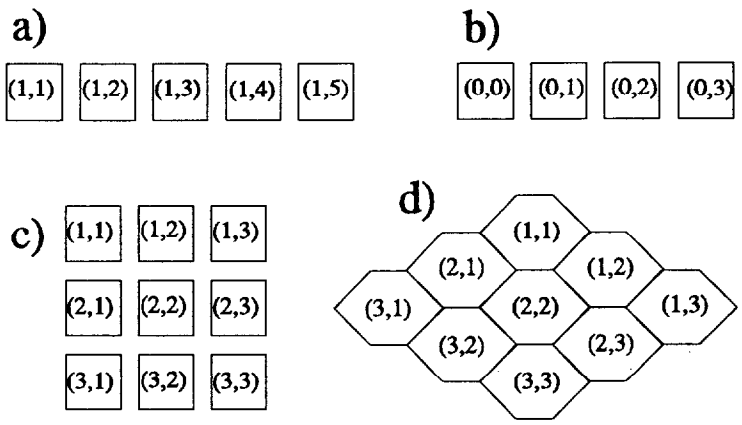

Fig. 5.2 Tipos de arrays permitidos.

Téngase en cuenta que los rangos que se han de especificar ahora indican las filas afectadas, primero, y las columnas, después. Las células situadas tomarán el nombre dado por la palabra '*cell*' y los índices de la posición en que se coloquen. Por ejemplo, '*cell(1,2)*' para la célula que se sitúe en la primera fila, segunda columna.

En el caso anterior, para acceder al pin *w* de la célula del tipo *Cell_1* situada en la posición x=2 e y=3, tendríamos que referenciarla con el siguiente path:

Sheet_1.array.cell(2,3).inp

Y de igual forma para acceder al pin inp de la célula del tipo Cell_2 que se ha instalado, hay que escribir:

Sheet_1.array.cell(0,0).a

AUTO:

Señala aquellos tipos de célula cuyos símbolos se desea que sean generados automáticamente por la herramienta. Su sintaxis es como sigue:

AUTO { nom_cel_1; nom_cel_2; ... }

donde con *nom_cel_x* se referencia los nombres de tipos de células.
Por ejemplo, para indicar que se genere automáticamente el símbolo para las células '*cel_A*', '*cel_B*' y '*cel_C*', ha de incluirse la línea:

AUTO { cel_A; cel_B; cel_C }

BLOCK:

El comando BLOCK permite realizar agrupaciones de elementos como células, arrays o elementos gráficos y conexiones entre ellos. Los blocks no tienen entradas salidas específicas como las células, sino que se comportan como macros.

Todas las coordenadas que se utilicen para el posicionamiento de elementos dentro de un bloque están referidas al origen del bloque. Los bloques permitirán realizar una librería de símbolos a partir de los elementos básicos que disponemos.

```
BLOCK nombre_block {
        PLACE { .... }
        CONNECT { .... }
        DRAWS { .... }
}
```

CELL:

Este comando permite definir las células básicas que componen un diseño. A continuación del comando CELL hemos de indicar el nombre asociado a la célula. Dentro de la definición de la célula pueden haber distintos campos de información que estarán dentro de otros tantos comandos. Todos los comandos son opcionales y pueden estar situados en cualquier orden.

```
CELL nombre_cell {
        INOUT { ... }
        INTERNAL { ... }
        FUNCTION { ... }
        MODEL { ... }
        DRAWS { ... }
          ...
}
```

En el comando INOUT se declararán las señales de entrada y salida de la célula. En INTERNAL se declararán las señales que vamos a utilizar internamente en la descripción de la funcionalidad de la célula. FUNCTION define la funcionalidad de la célula en distintos lenguajes. MODEL asocia a la célula diversos modelos de visualización. DRAWS en el nombre genérico que agrupa a distintos comandos que permiten añadir a la célula distintos elementos gráficos.

CONNECT:

Establece conexiones entre señales de un mismo sheet o de sheets diferentes. Debe especificarse el nombre completo de cada señal a conectar (bien un pin o una señal interna), es decir, toda la jerarquía de elementos que contienen a las células a las cuales pertenecen, separados por un carácter punto decimal ('.'). Opcionalmente, el diseñador puede especificar un nombre con el que referirse a ese nodo. Su sintaxis presenta varios casos que podemos agruparlos en dos grupos; conexiones señal a señal y bucles de conexión. Cada uno de estos casos los trataremos por separado.

Conexiones del tipo señal a señal:

Tienen la siguiente sintaxis:

```
CONNECT {
        ...
        señal_1 = señal_2  =  nom_conex ;
        ...
}
```

donde:

señal_1	Nombre completo de la primera señal a conectar.
señal_2	Nombre completo de la segunda señal a conectar.
nom_conex	Nombre lógico opcional a dar al punto de interconexión.

Por ejemplo, si se desea conectar el pin '*suma_in*' de la célula '*Cell_1*', y el pin '*suma_out*' de la célula '*cell(0,0)*' situada en el array '*Array_1*', dando el nombre lógico '*suma*' al punto de interconexión, se usará la línea:

```
CONNECT {
        Cell_1.suma_in = Array_1.cell(0,0).suma_out  =  suma ;
}
```

Conexiones del tipo bucle de conexión:

Dentro de las sentencias de conexión, los posibles casos que se pueden dar son los siguientes:

nombre_cell.nombre_señal;
nombre_array[indicador_fila][indicador_columna].nombre_señal

Al evaluar las expresiones de conectividad, hay que tener en cuenta cómo se evalúan éstas. Los índice siempre hay que ponerlos de menor a mayor. Primero se evalúa el lado izquierdo de la expresión y luego el derecho. Los índice que se expresan con ":" se evalúan primero y luego los que utilizan "*". Para que la expresión sea correcta, el número de conexiones que se realizan en ambas partes de la sentencia han de ser iguales.

Caso 1:

```
CONNECT {
        nombre_array[a:b][c*d] = nombre_señal;
}
```

esta sentencia se puede interpretar de la siguiente forma:

```
for (i = a; i <= b; i++) {
        for (j = c; j <= d; j++ ) {
                nombre_array[i][j] = nombre_señal;
        }
}
```

Caso 2:

En el caso de que la conexión se realice entre un array y un vector de datos la forma de operar es semejante.

```
CONNECT {
        nombre_array[a:b][c*d] = nombre_señal[e*f];
}
```

esta sentencia se puede interpretar como:

```
for (i = a; i <= b; i++) {
        k = e;
        for (j = c; j <= d; j++ ) {
                nombre_array[i][j] = nombre_señal[k++];
        }
}
```

Caso 3:

En este otro caso vemos

```
CONNECT {
        nombre_array[a:b][c*d] = nombre_señal[e:f];
}
```

esta sentencia se puede interpretar como:

```
k = e;
for (i = a; i <= b; i++) {
        for (j = c; j <= d; j++ ) {
                nombre_array[i][j] = nombre_señal[k];
        }
        k++;
}
```

DRAWS: TEXT, LIN, REC, BOX, POL, CIR ...

Bajo el nombre genérico de DRAWS se indica una serie de elementos gráficos de dibujo que van a ayudar a la visualización de las arquitecturas definidas. Los elementos gráficos que pueden ser dibujados se encuentran en la Tabla V.1. Tanto los elementos gráficos que se definen con los comandos DRAW, así como los que se generan automáticamente a partir de las descripciones de células, se dibujan en diferentes capas en función de su tipo. Los nombres asignados a cada capa se encuentran en la Tabla V.2. Los colores de las capas se definirán en un fichero de sintaxis LDV. La sintaxis de estos comandos es como sigue:

com_pas { <lista_par> }

donde:

Tabla V.1 Elementos gráficos de dibujo.

COMANDO	DESCRIPCION	PARAMETROS
TEXT	Dibuja un string de texto.	texto, capa, x, y
LIN	Dibuja una línea.	capa, x1, y1, x2, y2
REC	Dibuja un rectángulo.	capa, x1, y1, x2, y2
BOX	Dibuja una caja.	capa contorno, capa fondo, x1, y1, x2, y2
POL	Dibuja un polígono cerrado.	capa contorno, capa fondo, x1, y1, [xi, yi], ...
PLL	Dibuja una polilínea abierta.	capa contorno, x1, y1, [xi, yi], ...
CIR	Dibuja un círculo.	capa contorno, capa fondo, x1, y1, radio
LABEL	Asocia una etiqueta a una variable de una célula. El valor visualizado se refresca según cambie la variable.	capa, escala, x, y, variable
VOL	Asocia un indicador análogico de aguja (tipo voltímetro) a una variable de célula.	capa, escala, x, y, valor min, valor max, variable
SCROLL	Asocia una barra de scroll a una variable de una célula.	capa, escala, x, y, valor min, valor max, variable

Tabla V.2 Nombres asignados a las distintas capas.

NOMBRE	DESCRIPCION
L_BACKGR	Background.
L_BODYCEL	Cuerpo de un célula.
L_CONTCEL	Contorno de una célula.
L_PINCEL	Pines de una célula.
L_CONWIRE	Conexiones tipo wire.
L_CONBUS	Conexiones tipo bus.
L_TXTCEL	Texto de una célula.
L_TXTARR	Texto de un array.
L_TXTBLO	Texto de un bloque.
L_TXTDES	Texto de un Sheet.

com_pas Comando de dibujo de elemento pasivo.

<lista_par> Lista de parámetros propia de cada comando.

Por ejemplo, para mostrar la cadena de texto *'C.M.A.'* en las coordenadas (2,10) del elemento actual, con color L_TXTCEL, se pondría:

TEXT *{ L_TXTCEL 'C.M.A.' 2 10 }*

FUNCTION:

Envía los datos necesarios hacia el fichero asociado al simulador que se indica (deberá haber sido asociado previamente con un comando ***SIMUL***). Admite dos sintaxis posibles:

FUNCTION nom_sim { ... }

donde:

nom_sim Nombre del simulador al que se dirigen los datos.

O bien, la sintaxis:

FUNCTION nom_sim = 'nom_fich';

donde:

nom_sim Nombre del simulador al que se dirigen los datos.

nom_fich Nombre del fichero que contiene la descripción del elemento dada.

Con la primera sintaxis, la propia descripción del elemento se incluye entre las llaves del comando, de forma que siga plenamente la sintaxis propia del simulador al que va dirigida. Mientras que, en la segunda sintaxis, lo que se indica es el fichero donde puede encontrarse esa misma descripción.

Por ejemplo, si desea enviarse al simulador VHDL la descripción de un elemento interno, que se ecuentra en el fichero *'cel_A.vhdl'*, se habría de incluir la siguiente línea:

FUNCTION VHDL = 'cel_A.vhdl';

Comandos de descripción de netlist dentro de FUNCTION.

En los caso que no se desee realizar una descripción de comportamiento de las células de nuestros diseños, es posible recurrir a una descripción tipo netlist de su funcionalidad. Esta netlist será representada en los ficheros de salidas para los simuladores de forma apropiada. Esta descripción se realizará mediante un conjunto de sentencias, que estarán escritas entre las dos llaves del comando FUNCTION. Entendemos como sentencia toda aquella descripción de conectividad delimitada por ";". Existen sólo dos tipos de sentencias básicas, a las cuales nos referiremos como sentencias del tipo A y B. A lo largo de la exposición veremos como podemos escribir sentencias tan complicadas como queramos, utilizando éstas como bases. Para poder utilizar una primitiva dentro de este tipo de descripción debe haber sido declarada como célula anteriormente.

Ejemplo de los dos tipos de sentencias:

```
FUNCTION {
        (A) salida = PRIMITIVA(entrada1, ..., entradan);
        (B) PRIMITIVA(entrada1, ..., entradan)(salida1, ..., salidam);
}
```

Las sentencias del tipo A, pueden ser utilizadas con todas las primitivas lógicas con

una única salida. Las sentencias del tipo B, pueden ser utilizadas con cualquier tipo de célula independientemente del número de salidas que posea.

Todos los nombres de señales que utilicemos deben de estar previamente declarados como entradas, salidas o señales internas. Al escribir una señal, podemos emplear mayúsculas o minúsculas independientemente de como se haya declarado ésta. Los nombre de primitivas pueden ser también escritos en mayúsculas o minúsculas indistintamente. Es de resaltar, por lo tanto, que los nombres de las señales no deben coincidir con los de las primitivas empleadas.

El circuito más elemental que podamos diseñar, consiste en un simple inversor en el cual su entrada y salida, son también entradas y salidas de la celula. Para su definición, utilizaremos una única sentencia del tipo A.

```
INOUT { entrada:I, salida:O }
FUNCTION {
        salida = INV(entrada);
}
```

Supongamos ahora que tenemos la célula de la Fig. 5.3.

```
INOUT { inp[3:0]:I, salida:O }
INTERNAL  { sig[2:0] }
FUNCTION {
        sig0 = NAND2(inp0, inp1);
        sig1 = NOR2(inp2, inp3);
        sig2 = NAND2(sig0, sig1);
        out  = INV(sig2);
}
```

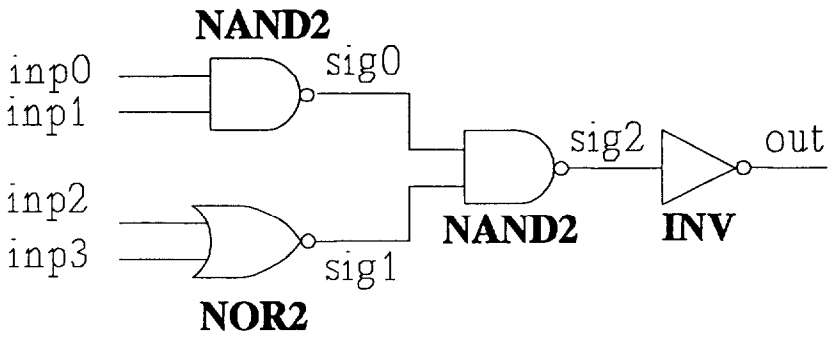

Fig. 5.3 Ejemplo de descripción de una netlist dentro de FUNCTION.

Existe otra forma más rápida de definir el circuito anterior. Como normalmente muchas de las señales internas del bloque no nos interesan asignarles un nombre concreto, podemos asignar directamente a la entrada de una primitiva la salida de otra. A esta operación la denominaremos asignación automática de salidas. El programa detectará que a la entrada de una primitiva, no hay señal, sino el nombre de otra primitiva, por lo que automáticamente generará un nombre a esa señal de salida de la primitiva.

```
INOUT  { inp[3:0]:I, out:O }
FUNCTION {
        out = INV(NAND2(NAND2(inp0,inp1),OR2(inp2,inp3)));
}
```

INOUT:

Especifica las características de los pines de E/S de la célula en definición (sólo puede usarse dentro de un comando ***CELL***). La información aportada dentro de este comando se utilizará para generar de forma automática el símbolo de la células. También se utilizará esta información para realizar las autoconexiones que se produzcan al definir un array. La sintaxis de este comando será como sigue:

```
INOUT {
        nombre_pin[a:b] : flujo : var ;
        .....
}
```

donde:

nombre_pin Nombre a dar al pin.

[a:b] Rango de líneas que forman ese pin. Es opcional y, si no se especifica, se tomará como [1:1] (es decir, una sola línea para el pin).

flujo Sentido y posición del pin, indicado mediante un nemónico. La primera de estas letras solo puede ser "I", "O", "B" o "G" que indican "Input", "Output", "Bidirectional" o "Global" respectivamente, las siguientes indican su situación geográfica de conexión "N", "S", "E", "W", "NE", "NW", "SE", "SW" (Fig. 5.4).

var Tipo de la variable asociada a este pin, codificado mediante un nemónico. Es opcional y, si no se especifica, se tomará un tipo BIT. Los posibles valores son los siguientes: BIT, BOOLEAN, LOG, CHAR, FLOAT, DOUBLE, INT8, INT16, INT32, UINT8, UINT16, UINT16.

Ejemplo:

```
INOUT {
        nombre_señal:IN,
        nombre_señal:OSE:FLOAT,
        nombre_bus[15:0]:OS,
        nombre_bus[15:0]:INE:FLOAT
}
```

Si se desea establecer la autoconexión entre dos pines de la célula, deberá usarse el operador '>'. Este operador se situará entre la definición de los dos pines, sustituyendo al ';' del final de la línea del primero de ellos. Por ejemplo, si se desea definir un pin de nombre '*suma_in*', que entra por el Noroeste, su variable es lógica y con 1 hilo, y autoconectarlo con el pin '*suma_out*', salida por el Sureste, tipo lógico y de 1 hilo,

se habrían de especificar las siguientes líneas:

INOUT {

....

suma_in : INW : LOG > suma_out : OSE : LOG;

....

}

Ejemplo:

INOUT {
a_input:IN > a_output:OS,
b_input:IE > b_output:OW,
c_input:IW
}

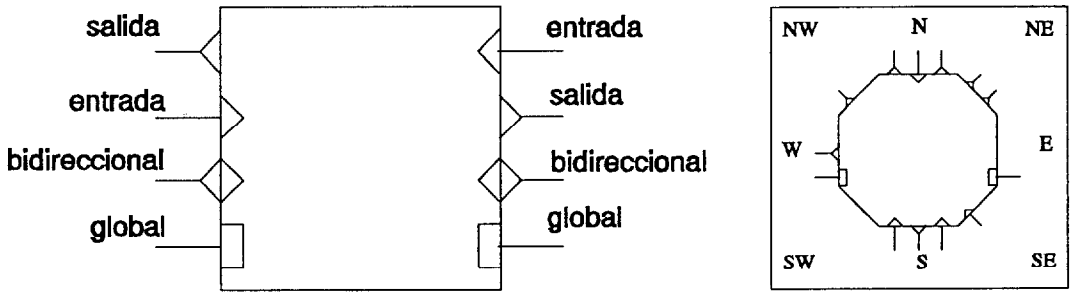

Fig. 5.4 Tipos y situación geográfica de pines.

INTERNAL:

Las señales internas definen las señales o variables de una célula que no pertenencen a su periferia. La sintaxis de este comando es similar a la del comando INTOUT, si bien hay parámetros que no precisa.

INTERNAL {

...

pin_int[a:b] : var ;

...

}

donde:

pin_int Nombre del pin interno que se define.

[a:b] Rango de líneas que contiene el pin interno. Es opcional y, caso de no especificarse, se tomará como valor [1:1].

var Tipo de la variable que representa este pin. Es opcional y, si no se especifica, se tomará BIT por defecto.

Por ejemplo, si se define un pin interno de nombre '*carry*', y de tipo BOOLEAN, se usará la línea:

INTERNAL *{ carry : BOOLEAN }*

Ejemplo:

INTERNAL *{ a, b, bus_a[0:7], bus_b[7:0] }*

PLACE:

Posiciona un elemento dentro de otro. No puede usarse dentro de una célula, ya que ésta es el elemento básico. Se usa tanto si se trata de poner un elemento en la **definición** de otro (array o bloque), como si se posiciona un elemento dentro de la **descripción** de un sheet.

La sintaxis de este comando es como sigue:

PLACE *{*

...

nom_log = nom_tipo : [x,y] : [ax,ay] : rot ;

...

}

donde:

nom_log Nombre lógico que se da al elemento que se posiciona.

nom_tipo Nombre del tipo de elemento a posicionar.

[x,y] Coordenadas del punto donde se situará el elemento, relativas al origen del elemento en que se sitúa.

[ax,ay] Ancho inicial opcional para las dimensiones con que se visualizará el elemento. Se recomienda un valor 100 o mayor. Si no se especifica, se tomará por defecto el valor 1000.

rot Angulo de rotación con que se situará el elemento al ser posicionado.

Ejemplo:

PLACE *{*

cel_a = Celula_1:[0,0]:[100,100]:0;

cel_b = Celula_1:[0,100]:[200,100]:90;

array_a = Array_1:[100,0]:[1000,1000]:0;

block_a = Block_1:[0,400]:[100,100]:0;

}

El nombre lógico que se asocia a un elemento (célula, array o bloque) en el momento de realizar el PLACE sobre otro elemento, es el que se utilizará para acceder a las variables de los distintos elementos.

Ejemplo:

```
CELL Cell_1 {
        INTOUT { inp:IN > out:OS }
}
BLOCK Block_1 {
        PLACE {
                cell = Cell_1:[0,0]:[100,100]:0;
        }
}
SHEET Sheet_1 {
        PLACE {
                block = Block_1:[0,0]:[100,100]:0;
        }
}
```

En el ejemplo anterior el camino completo para acceder a la varible *inp* de la célula posiciona dentro del bloque *Block_1* que se encuentra en la hoja *Sheet_1* será:

Sheet_1.block.cell.inp

MODEL:

Asocia un modelo de visualización con el elemento que se encuentra actualmente en definición. Los modelos de visualización disponibles se encuentran en la Tabla V.3. La sintaxis a emplear es la siguiente:

```
MODEL {
        ...
        nom_mod = 'desc' : 'nom_fich';
        ...
}
```

donde:

nom_mod Nombre del tipo de modelo a emplear.

'desc' Descripción genérica.

'nom_fich' Nombre opcional del fichero donde se encuentra el modelo.

Así, por ejemplo, para definir un modelo de VHDL para el elemento *'cel_A'*, tendremos una llamada a la función de la forma:

```
MODEL {
        V_VHDL = 'Descripción VHDL' : 'cel_A.desc' ;
}
```

donde el comentario y el nombre de fichero son simplemente a título de ejemplo.

Tabla V.3: Modelos reconocidos.

NOMBRE	MODELO DE...
V_CHDL	Descripción funcional de la célula en C
V_WAVES	Descripción de los vectores de simulación
V_LAYOUT	Layout de una célula en CIF
V_VERILOG	Modelo VERILOG de simulación
V_HILO	Modelo SYSTEM HILO de simulación
V_VHDL	Modelo VHDL de simulación
V_SIMON	Modelo para lenguaje LIV de SiMon
V_SHEET	Modelo para SHEET
V_PINEXT	Modelo para pin externo
V_PININT	Modelo para pin interno

SHEET:

Este comando nos permite definir la estructura de mayor nivel jerárquico, que es un diseño. A continuación del comando SHEET hemos de indicar el nombre que se le va a asociar al diseño. Este debe cumplir las mismas normas que se aplican al nombre de una célula.

Entre los paréntesis de comienzo y final de este comando, tendremos que posicionar los distintos elementos de la hoja. Mediante el comando CONNECT realizaremos las conexiones oportunas entre las distintas señales de las células. Al igual que en el resto de los elementos de dibujo, podemos utilizar las distintas instrucciones de posicionamiento de elementos gráficos para completar la vista de la hoja.

La especificación de un SHEET es totalmente semejante a la de un BLOCK con la salvedad que un SHEET no puede ser invocado desde otro SHEET. Además cada SHEET tendrá asociada una ventana gráfica en SiMon para su visualización.

```
SHEET nombre_sheet {
        PLACE { .... }
        CONNECT { .... }
        DRAWS { .... }
}
```

SIMUL:

Asocia los datos destinados a un simulador indicado por su nombre con un fichero de salida, de forma que cualquier salida de datos destinada a ese simulador sea dirigida y recogida por el fichero indicado. Los nombres de simuladores que actualmente acepta el LDAP vienen dados en la Tabla V.4.

Debe ser usado externamente a la definición de cualquier elemento, si bien puede hacerse en cualquier punto del fichero descripción y tantas veces como se desee. Por supuesto, cada vez que se utilice para un mismo simulador las salidas subsiguientes para ese simulador serán destinadas al último de los ficheros especificados para el mismo.

El nombre del fichero puede llevar o no extensión. Pero, si no la lleva, le será añadida por el programa automáticamente de acuerdo al tipo de simulador indicado.

La sintaxis será como sigue:

SIMUL nom_sim = 'nom_fich' ;

donde:

nom_sim Nombre del simulador al que se enviarán datos.

nom_fich Nombre del fichero que recoja los datos a enviar al simulador.

Por ejemplo, se desea que los datos preparados para el simulador de VHDL sean destinados a un fichero de nombre '*test.vhdl*' y, para el simulador de Verilog, el fichero '*test.verilog*'. Las líneas de orden serán:

SIMUL V_VHDL = 'test.vhdl';
SIMUL V_VERILOG = 'test.verilog';

Tabla V.4: Relación de las extensiones a usar para los ficheros de salida para simuladores

SIMULADOR	EXTENSION
CHDL	.chdl
WAVES	.waves
LAYOUT	.layout
VERILOG	.verilog
HILO	.hilo
VHDL	.vhdl
SIMON	.liv

USE:

Indica aquel fichero que desee señalarse para su inclusión en la descripción gráfica del sistema con el lenguaje LDV. Su sintaxis es como sigue:

USE nom_fich;

donde:

nom_fich Nombre del fichero que ha de indicarse como incluído.

Por ejemplo, para que el visualizador **SiMon** incluya el fichero '*test.ldv*' en se análisis, deberemos incluir la línea:

USE test.ldv ;

5.2 Lenguaje LDAP compilado.

El lenguaje LDAP compilado proporciona al usuario la posibilidad de realizar la descripción de un sistema mediante un conjunto de macrofunciones escritas en lenguaje C y desarrolladas con este propósito. La entrada de datos de la descripción se realiza en la forma de fichero fuente C, el cual deberá seguir un formato mínimo para que el LDAP pueda reconocer la información contenida en él. Este formato, bastante sencillo como se muestra a continuación.

```
....
....
#include <stdio.h>
#include "basico.h"

Main_Sheet() {

        Init_System();   /* Función de inicialización del sistema */

        ....
        ....
}
```

Según puede observarse en este listado, el fichero confeccionado por el diseñador debe comenzar por la inclusión del fichero ***basico.h***. Esto es fundamental, ya que en este fichero se contienen todas las definiciones de tipos de datos de que dispondrá aquél para llamar a las diferentes macrofunciones del lenguaje. Sin este fichero, no se podrá compilar correctamente la descripción dada por el diseñador.

Main_Sheet();

Esta es la función principal llamada por la librería aportada por el LDAP en el momento de arrancar el programa generado a partir de la compilación del fichero fuente. No se podrá realizar correctamente la compilación de éste si el diseñador no ha declarado una función con este nombre.

El fichero fuente que crea el diseñador con el lenguaje LDAP, debe poder ser compilado con las librerías aportadas por esta herramienta. La función ***Main_Sheet()*** deberá ser incluida por el diseñador de forma que englobe la llamada a todas las macrofunciones que emplee para la descripción del sistema. De no hacerlo así, es decir, si las llamadas a las macrofunciones del LDAP no se realizan dentro de ésta, el fichero podrá ser igualmente compilado, pero su ejecución no tendrá ningún efecto, ya que la llamada a la función ***Main_Sheet()*** se realiza con la suposición de que dentro de ella se hará la llamada a las funciones de generación del sistema.

Dentro de esta función, además de las macrofunciones del LDAP, se podrán usar todos los recursos de programación del C, de forma que el diseñador tiene libertad completa para realizar un completo programa C. Todo ello siempre y cuando se incluya como parte de la función ***Main_Sheet()***.

5.2.1 Macrofunciones del LDAP compilado.

Veamos a continuación la descripción de la sintaxis y parámetros a emplear en la llamada a las diferentes funciones del lenguaje LDAP. Para cada una de ellas se dará una descripción completa de su efecto, forma de llamada y parámetros posibles a emplear.

void Init_System();

Debe ser, ineludiblemente, la primera de las funciones que llame el diseñador en el fichero de descripción del sistema. Su efecto es el de inicializar los parámetros del sistema que hayan de ser usados y crear la estructura base de memoria a partir de la cual va a depender el resto de la información del sistema aportada por el diseñador. No precisa de ningún parámetro.

CELL *Init_Cell(char *name);
ARRAY *Init_Array(char *name);
BLOCK *Init_Block(char *name);
SHEET *Init_Sheet(char *name);

Cada una de estas funciones se utiliza para realizar la inicialización de la definición de cada uno de los cuatro tipos básicos de elementos que reconoce el LDAP. No se pueden usar hasta que la llamada anterior a cualquiera de ellas (si la hubo) sea cerrada con una función ***End_xxx()***. El único parámetro que llevan es el del nombre con que el usuario quiere identificar al tipo de elemento que se inicia.

End_Cell();
End_Array();
End_Block();
End_Sheet();

Cada una de estas funciones se utiliza para realizar el cierre de la definición de cada uno de los cuatro tipos básicos de elementos que reconoce el LDAP. No se puede usar ninguna de ellas si no se ha hecho, antes, una llamada a su función ***Init_xxx*** correspondiente. Por ejemplo, para finalizar la definición de un tipo de célula, se procede a emplear la siguiente línea:

End_Cell();

Def_Pin();

Define las características de un pin de la célula que esté actualmente en definición (debe usarse entre un comando ***Init_Cell("...")*** y su ***End_Cell()*** de cierre). Los parámetros que precisa son, por este orden, los siguientes:

* nombre que se da al pin,
* flujo de la señal asociada al pin (incluye lado de la célula donde se sitúa el pin y sentido de la señal),
* tipo de la variable asociada al pin, y
* número de hilos del pin (si más de 1, es un bus).

La información de flujo se proporciona mediante unos nemónicos definidos en ***basico.h*** y que se pueden ver en la XXX y XXX. Hay un nemónino especial para el flujo que recibe el nombre de STATE. Si este es el especificado, se asumirá que el

pin que se define es de tipo interno.

De igual forma, la información del tipo de variable se proporciona mediante los nemónicos que se aprecian en **XXX**. Por ejemplo, si se desea definir un pin de nombre '*suma_in*', que entra por el Noroeste, su variable es lógica y con 1 hilo, se habría de especificar la línea:

Def_Pin("suma_in", INW, BIT, 1);

Def_Con();

Define aquellos pines que han de autoconectarse en la célula que esté actualmente en definición (debe usarse entre un comando ***Init_Cell("...")*** y su ***End_Cell()*** de cierre). Los dos parámetros que lleva son los nombres de los pines que se van a conectar. Por ejemplo, para señalar para autoconexión los pines '*suma_in*' y '*suma_out*' de la célula actualmente en definición, se pondría la línea:

Def_Con("suma_in", "suma_out");

Def_Rec();

Sitúa células en el array actualmente en definición (debe usarse entre un comando ***Init_Array("...")*** y su ***End_Array()*** de cierre). Las células se señalan por el nombre de **su tipo**. No es necesario dar un nombre lógico a cada una de ellas, ya que de ello se encarga la propia función ***Def_Rec*** a medida que las va situando. El nombre que le asigna a cada una de ellas será de la siguiente forma:

cell(i,j)

donde: *i* es el número de la fila que ocupa, y *j* es la columna. La palabra '*cell*' será común a todas ellas y para todos los arrays.
Los parámetros que usa son los índices de la esquina superior-izquierda e inferior-derecha del rectángulo imaginario que cubrirán las células afectadas.

Por ejemplo, para situar 6 células del tipo '*Celula_1*' en la segunda y tercera fila, cuarta a sexta columna, del array en definición:

Def_Rec(2, 4, 3, 6, "Celula_1");

Place();

Posiciona un elemento dentro del que esté actualmente en definición y, por tanto, no puede usarse cuando éste sea una célula. Se usa tanto si se trata de poner un elemento en la **definición** de otro (array o bloque), como si se posiciona un elemento dentro de la **descripción** de un sheet. Los parámetros que precisa dan completa información del lugar y características de posicionamiento:

* coordenadas (x,y) donde situar el elemento (referidas al origen del elemento en que se sitúan),
* rotación a dar al elemento cuando se posicione (±360°),
* ancho inicial en ambos ejes,
* nombre lógico a darle a su posicionamiento, y

* tipo de elemento.

Por ejemplo, si se desea posicionar una célula del tipo '*Celula_1*', con el nombre '*cel_A*', en el punto (2,4), con tamaño (10,10) y 0° de rotación, se usaría la línea:

Place(2, 4, 0, 10, 10, "cel_A", "Celula_1");

Connect();

Realiza la conexión entre dos pines del sheet actualmente en descripción (debe usarse entre un comando ***Init_Sheet("...")*** y su ***End_Sheet()*** de cierre). Los dos parámetros que precisa son los nombres **completos** de los pines a conectar. Decimos completos porque deben especificar toda la jerarquía de elementos que contienen a las células a las que pertenecen esos pines, a excepción del propio nombre del sheet. Opcionalmente, puede especificarse un nombre con el que referirse a ese nodo.

Por ejemplo, si se desea conectar el pin '*suma_in*' de la célula '*cel_A*', y el pin '*suma_out*' de la célula '*cell(0,0)*' situada en el array '*arr_C*', dando el nombre '*suma*' al punto de interconexión, se usará la línea:

Connect("cel_A.suma_in", "arr_C.cell(0,0).suma_out", "suma");

Def_Simul();

Asocia los datos destinados a un simulador indicado por su nombre con un fichero de salida, de forma que cualquier salida de datos destinada a ese simulador sea dirigida y recogida por el fichero indicado. Los nombres de simuladores que actualmente acepta el LDAP vienen en el apartado 5.1.

Debe ser usado externamente a la definición de cualquier elemento, si bien puede hacerse en cualquier punto del fichero descripción y tantas veces como se desee. Por supuesto, cada vez que se utilice para un mismo simulador las salidas subsiguientes para ese simulador serán destinadas al último de los ficheros especificados para el mismo.
El nombre del fichero puede llevar o no extensión. Pero, si no la lleva, le será añadida por el programa automáticamente de acuerdo al simulador indicado.

Por ejemplo, se desea que los datos preparados para el simulador de VHDL sean destinados a un fichero de nombre '*test.vhdl*' y, para el simulador de Verilog, el fichero '*test.verilog*' . Las líneas de orden serán:

Def_Simul(V_VHDL, "test.vhdl");
Def_Simul(V_VERILOG, "test");

Como puede observarse, no se ha puesto extensión al fichero de Verilog. El fichero de salida se llamará, por tanto, '*test.verilog*'.

Function();

Envía los datos de salida, para el simulador indicado, al fichero que haya sido previamente asociado con él mediante un comando ***Def_Simul()*** apropiado. Debe

usarse durante la definición de un elemento. Los datos a enviar al simulador serán tomados del fichero indicado en la llamada a ***Function()***. Este fichero puede contener, según el primer parámetro indicado, los siguientes datos:

* la descripción interna del elemento al que se asocia, de acuerdo con la sintaxis propia del simulador al que vaya destinado, o bien,
* datos para el programa monitor **SiMon** en el lenguaje LIV (*Lenguaje de Intercambio de Variables*), en cuyo caso, el primer parámetro habrá de ser V_SIMON.

Por ejemplo, si desea enviarse al simulador VHDL la descripción de un elemento interno, que se encuentra en el fichero '*cel_A.vhdl*', se habría de incluir la siguiente línea:

Function(V_VHDL, "cel_A.vhdl");

Def_Model();

Asocia un modelo de visualización con el elemento que se encuentra actualmente en definición. Los modelos de visualización disponibles están presentados en el apartado 5.1. La llamada a esta función lleva una serie de parámetros, tales como:

* tipo de modelo de visualización,
* información descriptiva del modelo (texto genérico), y
* nombre del fichero que contiene el modelo del elemento.

Así, por ejemplo, para definir un modelo de VHDL para el elemento '*cel_A*', tendremos una llamada a la función de la forma:

Def_Model(V_VHDL, "Descripción VHDL", "cel_A.desc");

donde el comentario y el nombre de fichero son simplemente a título de ejemplo.

Use();

Indica aquel fichero que desee señalarse para su inclusión en la descripción gráfica del sistema con el lenguaje LDV.
Por ejemplo, para que el visualizador **SiMon** incluya el fichero '*test.ldv*' en se análisis, deberemos incluir la línea:

Use("test.ldv");

AutoSymbol();

Señala aquellos tipos de célula cuyos símbolos se desea que sean generados automáticamente por la herramienta. Acepta un número variable de parámetros (todos ellos, nombres de tipos de célula), señalando el final de la lista de los mismos mediante el valor NULL. Por ejemplo, para indicar que se genere automáticamente el símbolo para las células '*cel_A*', '*cel_B*' y '*cel_C*', ha de incluirse la línea:

AutoSymbol("cel_A", "cel_B", "cel_C", NULL);

Draw_Pas();

Indica el dibujo de un elemento pasivo, tal como, línea, caja, rectángulo, círculo, texto, etc. . Acepta un número variable de parámetros de acuerdo al elemento pasivo de que se trate, señalando el final de la línea mediante el valor STOP.

En la llamada a esta función, el primer parámetro que se especifique debe ser uno de los nombres válidos de comando gráfico que explicamos en el apartado 5.1, de forma que la función sepa como interpretar los parámetros que le siguen. Estos serán, según el comando:

* Para LIN:
  - nivel de color para la línea,
  - coordenadas del punto inicial de la línea, y
  - coordenadas del punto final de la línea.
* Para REC:
  - nivel de color para las líneas del rectángulo,
  - coordenadas de la esquina inferior-izquierda, y
  - coordenadas de la esquina superior-derecha.
* Para BOX:
  - nivel de color para el contorno de la caja,
  - nivel de color para el fondo de la caja,
  - coordenadas de la esquina inferior-izquierda, y
  - coordenadas de la esquina superior-derecha.
* Para TEXT:
  - texto a representar,
  - nivel de color del texto, y
  - coordenadas del punto donde comenzar a escribir el texto.
* Para CIR:
  - nivel de color para la línea de la circunferencia,
  - coordenadas del centro, y
  - radio.
* Para POL:
  - nivel de color del contorno del polígono,
  - nivel de color del fondo del polígono,
  - lista de coordenadas de sus vértices.

Por ejemplo, para mostrar la cadena de texto '*C.M.A.*' en las coordenadas (2,10) del elemento actual, con color L_TXTCEL, se pondría:

Draw_Pas(TEXT, "C.M.A.", L_TXTCEL, 2, 10, STOP);

5.2.2 Salida generada por el LDAP compilado.

El programa ejecutable que se genera de la compilación del fichero fuente con el lenguaje LDAP compilado, llevará el nombre del fichero de entrada con la descripción del sistema, pero sin la extensión. La ejecución de este programa crea una serie de ficheros de salida todos ellos con el mismo nombre raíz que el fichero de entrada pero con una extensión acorde a su contenido.

Los ficheros así generados se diferenciarán de los generados por el intérprete mediante un carácter *underscore* que se antepone al nombre del fichero. De acuerdo con esto, para un fichero de entrada de nombre '*test.c*', se generarán los siguientes ficheros de salida:

_test.bin Contiene toda la información que generó en memoria la ejecución del programa del usuario, siendo una copia exacta de las estructuras de memoria.

_test.info Contiene una descripción textual del sistema descrito por el diseñador, con información relativa a la cantidad de memoria ocupada por cada uno de los elementos que constituyen el sistema.

_test.ldv Contiene la descripción gráfica del sistema basada en el LDV (*Lenguaje de Descripción Visual*). Este fichero será el que use como entrada el programa de monitorización **SiMon**.

5.3 Lenguaje de descripción gráfica (LDV).

El LDV es un lenguaje descriptivo jerarquizado que permite definir elementos simples que serán utilizados para la creación de otros más complejos.

Existen instrucciones que generan salida gráfica, salida textual e instrucciones que generan variables. Además, existen otras instrucciones las cuales son utilizadas para la definición de módulos elementales o primitivas que serán usadas en otras primitivas más complejas.

En el *LDV* ha de ir un comando por línea. Podrán incluirse líneas en blanco. Se tomará como finalizador de comando el retorno de carro. El formato de cada instrucción se describirá en detalle en los apartados que siguen a continuación. En general, diremos que cada instrucción comienza con una palabra clave seguida por una lista de parámetros. Se utilizarán como separadores cualquiera de los caracteres siguientes: espacio en blanco, tabulador, punto, punto y coma.

Existen tres elementos adicionales que se contemplan en el lenguaje de descripción visual. El primer elemento que se considera es la declaración de comentario (#), el segundo es la inclusión de archivos (*) y el último será el entrecomillado (" ").

1) En cuanto a la declaración de comentario diremos que cualquier texto que se escriba al final de la línea tras todos los parámetros de las instrucciones (excepto POL), serán considerados como comentarios e ignorados. A parte de éso, también serán considerados como comentarios todas las líneas cuyo primer carácter útil (es decir, descartando todos los separadores) sea una almohadilla (#).

2) En cuanto al asterisco (*), sirve para indicarnos que se pretende incluir un archivo. Si tenemos una línea cuyo primer carácter útil es el asterisco, sabemos que a través de ella se quiere incluir un archivo cuyo nombre irá después del asterisco. Puede llevar o no separadores entre el asterisco y el nombre del archivo que se pretende incluir. El archivo que se pretende incluir deberá tener declaraciones siguiendo el formato LDV. El efecto que se obtendrá será que el archivo incluido esté insertado en el punto en el cual sea invocado.

3) En lo que respecta al entrecomillado (" ") diremos que lo que se encuentre dentro de comillas será considerado un parámetro único. Esto nos será útil para incluir separadores dentro de los nombres de los elementos. Por lo tanto, el único carácter que no podrá ser incluido dentro del nombre de un elemento será la comilla doble.

5.3.1 Primitivas de documentación.

En este apartado se van a comentar las instrucciones *DATE, AUTHOR, VERSION, LIBRARY y MODEL.* Estas instrucciones son simples y contienen información textual que generalmente no será necesaria para la descripción. Esta información tan sólo tiene sentido en cuanto a la identificación de los archivos fuentes. No tienen ningún efecto en la representación gráfica, ni en las variables generadas ni en la jerarquía. Pueden ir en cualquier punto de la descripción del sistema.

DATE <fecha>

Esta instrucción almacena la fecha de generación del código. Es de carácter informativo y no tiene efecto sobre la descripción del sistema.

AUTHOR <autor>

Esta instrucción almacena el autor del código generado. Al igual que la anterior es de carácter informativo y no tiene efecto sobre la descripción del sistema.

VERSION <version>

Esta instrucción almacena la versión del código actual. No tiene efecto sobre la ejecución ni la interpretación del sistema.

LIBRARY <archivo_librería>

Esta instrucción referencia a un elemento de librería que puede ser usado en la descripción. En la práctica no tiene ningún efecto sobre la descripción y sólo es usado como información textual. Es, por tanto, de tipo informativo.

MODEL <tipo> [archivo]

Esta instrucción asocia un modelo al elemento en el que esté incluido. El campo tipo indica el tipo de modelo del que se trata. Puede ser: *CIF, VHDL* u otros. En general, puede ir cualquier combinación de caracteres. El sistema reconocerá algunos en concreto y a otros los ignorará. Opcionalmente puede ir el nombre de un archivo donde se encuentra el modelo indicado. Esta instrucción puede tener efecto sobre la descripción. Para algunos modelos en concreto se puede reclamar la descripción y se podría ver una representación de ella.

5.3.2 Primitivas gráficas.

En este apartado se van a describir las instrucciones que tienen salida gráfica. Estas instrucciones son *LEVEL, LIN, REC, BOX, POL, TEXT.* Estas instrucciones tienen representación gráfica efectiva que es generada por la herramienta cuando se encuentra con ellas. Dichas instrucciones no pueden estar declaradas como globales.

LEVEL <id> <R> <G> <B> <nombre> [pattern]

Esta instrucción nos permite definir niveles a los cuales se les asocia un color, un número, un nombre y un patrón de rellenado. El parámetro *id* nos identifica el número de nivel, el cual será referenciado por los elementos de dibujo para ser asociado al color y patrón de rellenado de éste. El número deberá estar en el intervalo (0,255). A continuación van tres parámetros que indican el color asociado a este nivel. El número *R* indica el nivel de rojo, el número *G* indica el nivel de verde y el *B* el de azul. Los niveles de cada color pueden variar entre 0 y 255, pudiendo formar con ellos hasta 16 millones de tonos. En la práctica, podemos especificar cualquier valor arbitrario de color y éste será tomado como valor para pintar todos los objetos que estén asociados a este nivel. El parámetro *nombre* almacena un nombre que puede servir también para identificar el nivel. En principio es sólo información textual. Por último, el parámetro *pattern* es opcional y especifica el nombre de un archivo que contiene una declaración de bitmap que será utilizada como patrón de rellenado. El rellenado será de tal forma que los bits a 1 serán pintados con el color asociado al nivel y los bits a 0 respetarán el fondo que tienen bajo ellos. En caso de no estar este parámetro declarado o no encontrarse el archivo al que se refiere, se utilizará un rellenado sólido.

El número de identificación a la hora de declarar el LEVEL es importante puesto que entre más bajo es el número menor prioridad tendrá en la representación. Esto quiere decir, que un número superior siempre será pintado sobre otro inferior, el cual puede quedar oscurecido o eliminado de la representación por éste.

LIN <nivel> <X0> <Y0> <X1> <Y1>

Esta instrucción dibuja una línea recta. El color de la línea será el asociado al nivel especificado en el parámetro *nivel*, dibujándose la línea siempre que dicho nivel esté activo. Las coordenadas entre las cuales será pintada la línea son *(X0,Y0)* y *(X1,Y1).*

REC <nivel> <X0> <Y0> <X1> <Y1>

Esta instrucción tiene un formato muy similar al comando anterior y dibuja un rectángulo tomando como esquinas las coordenadas *(X0,Y0)* y *(X1,Y1).* El color de ese rectángulo será el especificado por *nivel.*

BOX <fondo> <recuadro> <X0> <Y0> <X1> <Y1>

Esta instrucción dibuja un rectángulo relleno. Las coordenadas de dibujo de

este rectángulo viene indicada por *(X0,Y0)* y *(X1,Y1)*. El efecto de esta instrucción es pintar un rectángulo con el color asociado al nivel indicado por el parámetro *recuadro* y rellenar dicho rectángulo con el color y el pattern asociado con el nivel indicado por el parámetro *fondo*. Al estar asociado a dos niveles diferentes puede ocurrir que dependiendo del estado de los niveles (visible o no), pueda verse parcialmente o no verse el borde o el rellenado.

POL <fondo> <recuadro> <X0 Y0> [X1 Y1] ...

Esta instrucción dibuja un polígono relleno entre los puntos que se especifican en la declaración. El número de lados de este polígono es arbitrario siendo siempre mayor o igual que tres (si fuesen menor que tres esta instrucción no tendría efecto), teniendo que haber como mínimo un punto. El color del borde del polígono es el asociado con el nivel indicado por el parámetro *fondo* y se rellenará con el color y el pattern asociado con el nivel *recuadro*.

PLL <fondo> <recuadro> <X0 Y0> [X1 Y1]

Esta instrucción dibuja un polilínea abierta entre los puntos que se especifican en la declaración. El número de lados de este polilínea es arbitrario siendo siempre mayor o igual que dos (si fuesen menor que dos esta instrucción no tendría efecto), teniendo que haber como mínimo un punto. El color del borde del polígono es el asociado con el nivel indicado por el parámetro *fondo* y se rellenará con el color y el pattern asociado con el nivel *recuadro*.

TEXT <nivel> <escala> <X> <Y> <texto>

Esta instrucción escribe un *texto* del color asociado con el parámetro *nivel* y con un factor de escalado indicado por el parámetro *escala*, que debe ser entero. El valor uno es el valor mínimo de escalado utilizado para dibujar un texto. En principio, se puede poner como valor de escalado cualquier número arbitrario pero, si dicho número es excesivo, no tendrá efecto puesto que la herramienta tiene un tamaño máximo de letra. Los parámetro *X* e *Y* nos indican en que posición se colocará el texto que a su vez viene almacenado en el parámetro *texto*.

5.3.3 Declaración de variables.

En este apartado se van a describir las instrucciones que se van a utilizar para la declaración de variables y que son *INT, PIN y LABEL*. Estas instrucciones siempre tienen que ir dentro de la declaración de alguna primitiva, es decir, dentro de una célula (*CELL*), bloque (*BLOCK*) o diseño completo (*SHEET*), por tanto, nunca podrán ser declaraciones globales.

INT <nombre> <tipo> <nivel>

Esta instrucción define una variable interna. El nombre de esa variable viene indicado por el parámetro *nombre*, su tipo por el parámetro *tipo* y estará asociado con el nivel especificado por el parámetro *nivel*. Los tipos disponibles

son:

BIT:	define variables lógicas. Toman el valor de un bit.
BOOLEAN:	define variables lógicas. Toman el valor de un bit.
CHAR:	define variables tipo carácter. Toman valor de un carácter.
FLOAT:	define variables flotantes simples.
DOUBLE:	define variables flotantes de tamaño doble.
INT8:	define enteros de 8 bits.
INT16:	define enteros de 16 bits.
INT32:	define enteros de 32 bits.
UINT8:	define enteros sin signo de 8 bits.
UINT16:	define enteros sin signo de 16 bits.
UINT32:	define enteros sin signo de 32 bits.

PIN <nombre> <tipo> <tipo_pin> <nivel> <X> <Y>

Esta instrucción define una variable asociada a un PIN. El nombre de este PIN lo indica el parámetro *nombre*, el tipo de variable lo indica el parámetro *tipo* (corresponde a los tipos definidos anteriormente). El parámetro *tipo_pin* indica el tipo de PIN que estamos tratando. Este tipo puede ser:

IN, IE, IW, IS, INE, ISE, ISW, INW,
ON, OE, OW, OS, ONE, OSE, OSW, ONW,
BN, BE, BW, BS, BNE, BSE, BSW, BNW,
GN, GE, GW, GS, GNE, GSE, GSW, GNW.

Este tipo se refiere a si son de entrada, salida, si son por el norte, sur, noreste. El parámetro *nivel* especifica el nivel que tiene asociado este PIN. Las coordenadas *X* e *Y* nos indican en que punto del elemento está asociado.

LABEL <nivel> <escala> <X> <Y> <variable>

Esta instrucción nos define una etiqueta la cual contiene un nombre de variable y el valor de ésta. Cuando la variable varía, esta variación se reflejará dinámicamente en la representación. En la representación se podrá ver el nombre de variable seguido de su valor. La escala corresponde al valor asociado al parámetro *escala* y las coordenadas *X* e *Y* nos indicará en que punto será presentado el texto.

5.3.4 Definición de primitivas.

En este apartado se van a comentar las instrucciones *CELL, BLOCK, SHEET y END*. Estas instrucciones nos permiten definir primitivas para construir sistemas. Al igual que en los lenguajes de programación estructurados existen subrutinas aquí, podemos definir primitivas elementales para construir otras más complejas. En sí, existen tres tipos de primitivas elementales que son las tres primeras que hemos nombrado en este apartado. Así, *CELL* es la primitiva más elemental que se puede definir. No puede tener referencias a otra primitiva elemental. *BLOCK* es una primitiva de nivel intermedio. Esta primitiva puede tener referencias a otras primitivas de tipo *CELL* o *BLOCK* que hayan sido declaradas previamente.

Por último, *SHEET* es la primitiva de nivel jerárquico superior. En cuanto a declaración es idéntica a *BLOCK* salvo que *SHEET* es un elemento final. *BLOCK* es una primitiva que puede ser considerada como de paso o temporal. Es una primitiva de apoyo utilizada para declarar bloques más complejos. *SHEET* es un elemento final en tanto en cuanto es una declaración de algo real.

Para aclarar este concepto podemos hacer una similitud con el lenguaje de programación C. En él se definen primitivas que son las funciones pero la función main() es la función final y es la que se ejecutará. Podemos declarar funciones las cuales no utilicemos nunca pero, siempre que una función esté referenciada directa o indirectamente por main() será utilizada. De esta forma, podemos asemejar a SHEET con la función main(). Cada SHEET que exista en la declaración generará un ventana gráfica con un descripción y unas variables finales. Puede referenciar a BLOCK o CELL y tantos elementos como hagan falta.

CELL <nombre> <sizeX> <sizeY>

Esta instrucción define el inicio de la declaración de un *CELL*. El nombre del *CELL* que se define esta indicado por el parámetro *nombre* y el tamaño en unidades de dibujo viene dado por *sizeX* x *sizeY*. Este tamaño es importante a la hora de saber cual es el ámbito del *CELL* ya que cuando queramos seleccionar elementos que contiene un gráfico, podremos saber si nos encontramos dentro o fuera de algún *CELL* tomando como referencia estas coordenadas. Tras esta instrucción viene el cuerpo de declaraciones del *CELL* finalizando con la instrucción *END*.

BLOCK <nombre>

Esta instrucción define una primitiva de mayor nivel jerárquico que *CELL*, cuyo nombre viene especificado por el parámetro *nombre*. Puede contener *CELL* utilizando la instrucción *PLACE* que se explicará más adelante. De la misma forma puede contener otros *BLOCK* y *ARRAY*. Tras la instrucción *BLOCK* viene su cuerpo de declaraciones que finaliza con la instrucción *END*.

SHEET <nombre>

Esta instrucción es muy similar a *BLOCK* difiriendo de ella en que *SHEET* es un elemento final de declaración. Esto significa que *SHEET* genera una representación gráfica real y además genera variables reales. Todos los elementos que estén en la descripción de un *SHEET* son ubicados físicamente, reservándose para ellos memoria real que es asociada al *SHEET*. Tras el nombre de la instrucción viene su cuerpo de declaración que como en los casos anteriores finaliza con la instrucción *END*. El *SHEET* puede contener primitivas de menor nivel jerárquico, es decir, *CELL*, *BLOCK* o *ARRAY* utilizando para ello la instrucción *PLACE*.

END

Esta instrucción se utiliza para finalizar los cuerpos de declaraciones que existan en el lenguaje de descripción visual. Se ha visto su utilidad para finalizar los cuerpos de declaración de *CELL*, *BLOCK* o *SHEET* y ya se verá

más adelante finalizando la instrucción *NETLIST*.

5.3.4.1 Primitivas de posicionamiento.

En este apartado se van a tratar las primitivas de posicionamiento *PLACE y ARRAY*. Estas instrucciones se van a utilizar para referenciar elementos complejos que hayan sido ya declarados. El funcionamiento de *PLACE* es un poco diferente al de *ARRAY* en cuanto a posicionamiento.

PLACE <nombre> <X> <Y> <rot> <elemento> [SX] [SY]

Esta instrucción se utiliza para posicionar un elemento que haya sido declarado en el cuerpo de declaración de otro elemento. Es decir, en la práctica se utilizará *PLACE* para posicionar un *CELL*, *ARRAY* o *BLOCK* dentro de un *BLOCK* o *SHEET*. El *PLACE* tiene un *nombre* que será utilizado para identificarlo. Además tendrá una ubicación en unidades lógicas de dibujo las cuales vendrán dadas por los parámetros *X* e *Y*, que será tomadas como origen para el elemento que se posiciona. El elemento además, estará rotado sobre esa coordenada, tal como indique el parámetro *rot*, *rot* grados. El parámetro *elemento* nos indica el elemento que se va a posicionar. Opcionalmente se le puede añadir los tamaños *SX* y *SY*. Si se incluyen dichos parámetros, el elemento se posicionaría con un ancho *X* de valor *SX* y un ancho *Y* de valor *SY*. En caso de no incluirlos, el elemento es posicionado con su tamaño original. Los parámetros *SX* y *SY* nos pueden servir para hacer escalados diferentes a mismos elementos que se posicionan en diferentes ubicaciones. Diferentes *PLACE* generan diferentes posicionamientos que pueden ser del mismo elemento. Por ello, es importante que cada *PLACE* tenga un nombre diferente para poderlos diferenciar unos de otros. Esto es debido, a que el elemento que posicione *PLACE* puede contener variables las cuales, hay que diferenciar de las mismas variables generadas por otro *PLACE*. Por ello se debe utilizar el *nombre del PLACE* como parte del nombre que se utilice para localizar la variable.

ARRAY <nombre> <X> <Y> <col> <fil> <inc_col> <inc_fil> <elemento>

Esta instrucción nos permite replicar un elemento un número determinado de veces. El *ARRAY* tiene un *nombre* para poder identificarlo, que se le pasa en el parámetro *nombre*. Además tiene unas coordenadas de dibujo *X* e *Y* que corresponden con un desplazamiento inicial que se utilizará a la hora de dibujar el *ARRAY*. Los parámetros *col* y *fil* nos indicarán el número de columnas y filas que tiene el *ARRAY*. Además los parámetro *inc_col* y *inc_fil* nos indican el incremento gráfico que se ha de aplicar en columnas y filas a la hora de dibujar el elemento. El parámetro *elemento* nos da el nombre del elemento que se pretende replicar. Este elemento puede ser un *BLOCK* o un *CELL*. Esta primitiva no implica posicionamiento real del elemento al que se refiere. Para que un *ARRAY* esté realmente posicionado debe ser referenciado desde un *PLACE* que esté dentro de un *SHEET*.

5.3.4.2 Primitivas de Interconexión.

En este apartado se van a tratar las instrucciones que se utilizan para definir nodos, entendiendo por nodos a la agrupación de variables, la cual puede ser considerada como variable única. Las instrucciones que en concreto que vamos a definir son *NETLIST y NODE.*

NETLIST

Con esta instrucción iniciamos la declaración de una lista de nodos que acabará con la instrucción *END.* En principio se podrían declarar varias *NETLIST*, pero en la práctica se van a tratar todas como si fueran una, es decir, que todos los nodos que se definan se van a unificar en una *NETLIST* efectiva única. Esta instrucción debe ser declarada como global, nunca dentro del cuerpo de declaración de otra instrucción *CELL, BLOCK* o *SHEET.*

NODE <nombre>

Esta instrucción define un nodo o agrupación de variables que sólo puede encontrarse en el cuerpo de descripción de una instrucción *NETLIST.* Todas las variables que sean declaradas en el nodo serán declaradas como una variable única, de forma que un cambio en una de ellas afectará a todas las demás. El nombre del nodo será el indicado por el parámetro *nombre.* Este nombre será utilizado para referenciar a las variables que están agrupadas en él. Tras esta instrucción viene una lista de variables que finaliza con una declaración de otro nodo, es decir con otra instrucción *NODE,* o con una instrucción *END.* La instrucción *END* a parte de indicar el final de un nodo también indicará el final del *NETLIST* en el cual se encuentre insertada la instrucción *NODE.*

5.4 Lenguaje de intercambio de variables (LIV).

SiMon tras la lectura de una descripción *LDV* de un sistema posee una representación en memoria de éste. Esta descripción contendrá una serie de variables que deberán poder ser alteradas desde una fuente externa. Por ello, el programa deberá disponer de unas funciones de comunicación con el mundo externo, así, cualquier proceso externo usando estas funciones podrá comunicarse con nuestro proceso. De esta forma, el programa podrá conocer los nuevos valores de sus variables y reflejarlos de inmediato en la representación gráfica que está generando.

Esta comunicación es básica para cualquier programa de monitorización (representa la actualización de las variables que se va produciendo en tiempo real). Cada variable tiene un nombre lógico, por el que podrá ser referenciada. Las operaciones que se podrán realizar con estas variables serán básicamente asignaciones. Basándonos en sus nombres lógicos podremos actualiza variables determinadas, reflejando variaciones que se están produciendo en los datos del mundo exterior. Aparte de las operaciones de asignación de valores a variables existen otras para definir listas de variables para poder referenciar conjuntos de éstas e instrucciones para la gestión de la comunicación, etc.

De esta forma, podremos adaptar la entrada de nuestro programa a la salida de la mayoría de

los simuladores. Así, los comandos que se han implementado son los que siguen a continuación:

- *ASIGNACION SIMPLE;*
- *ASIGNACION MULTIPLE;*
- */LIST n;*
- */SET n;*
- */CLEAR n;*
- */END;*
- */PAUSE;*
- */WHAT;*
- */WHATIS variable;*
- */REFRESH win;*

Estos comandos *LIV* son los que se han de utilizar en la comunicación con *SiMon*. Para ello, el proceso externo que quiera comunicarse con *SiMon* tendrá que haber habilitado un canal de comunicación, por el que se le enviarán estos comandos, y a través del cual se obtendrán cambios de variables realizados en *SiMon*.

En la sintaxis de los comandos *LIV* se ignorarán todos los retorno de carro. Se tomarán como separadores el espacio en blanco, el tabulador, el punto y el punto y coma. Los separadores pueden ir en cualquier parte que no sea en medio de los nombres reservados de comandos, y en medio del texto de los parámetros. Se utilizará como finalizador de comando el punto y coma. Ningún comando será procesado hasta que no se encuentre su punto y coma finalizador. Se tratará de procesar los comandos cuando se encuentre un punto y coma. Los comandos erróneos serán ignorados y no provocarán error. En general los errores serán ignorados y no serán tenidos en cuenta.

5.4.1 Asignaciones de variables.

El primer tipo de comandos está destinado a asignaciones a variables. El caso más sencillo es la *ASIGNACION SIMPLE* en la cual se asigna un valor a una variable. En la *ASIGNACION MULTIPLE* se pasa una lista de valores que será asignada a las variables de una lista previamente asignada. El formato de la asignación simple es:

nombre_variable=valor;

y el de la asignación múltiple es:

valor1 valor2 valor3 ... valorn;

Cada valor será asignado a cada variable en el orden estricto en que fueron declaradas en la lista. En caso de que no hubiera igual número de variables en la lista que valores en la asignación múltiple, siempre se asignará el mínimo número, dicho de otro modo, si hubiesen más variables que valores, quedarían variables sin asignar y si hubiesen más valores que variables, habrían valores que no serían asignados.

El sistema, en general devolverá una asignación simple cada vez que sea alterada una variable desde dentro de el sistema. Esta variación será motivada por alguna operación que realice el usuario con el interfase gráfico. Es decir, que cuando el usuario de la herramienta, active

alguna variable que esté declarada en la descripción *LDV*, y altere su valor, está devolverá el nuevo valor, con formato de asignación simple.

5.4.2 Gestión de listas de variables.

El segundo tipo de comandos se refiere a la gestión de listas de variables. Así, por ejemplo:

/LIST n;

El comando */LIST* nos permite definir una lista de variables. El formato de este comando es el siguiente:

```
/LIST n;
        variable1;
        variable2;
                ...
        variable X;
/END;
```

Dicho comando es un creador de la lista que nos permitirá referenciarla una vez definida. El identificador *n* ha de ser un número entero. Es importante indicar que no podrán existir dos listas con el mismo identificador. El comando tiene asociado una serie de variables, una a continuación de otras hasta que se encuentre el comando */END* que indica el final del comando */LIST n*. Una vez declarada la lista podrá ser referenciada para hacer asignación a sus variables.

/CLEAR n;

El comando */CLEAR n* nos permite borrar listas previamente definidas. Utilizando ambos comandos, */LIST n* y */CLEAR n*, podemos mantener en memoria las listas que nos sean necesarias y eliminar listas las que no se quieran volver a utilizar. Si a este comando no se le especifican parámetros se borran todas las listas que estén en memoria.

/SET n;

El comando */SET n* sirve para acceder a las variables de una lista. El formato de este comando es el siguiente:

```
/SET n;
        asignación múltiple;
        asignación múltiple;
        asignación múltiple;
         ...
/END;
```

Todos los comandos deben ir acabados en punto y coma. En caso de que no hubiese punto y coma al final del mismo, éste no sería tenido en cuenta y se mezclará con el siguiente comando. En la asignación múltiple, por ejemplo, se tomará como separador, el espacio en blanco, el retorno de carro o el tabulador y podrán haber tantos como se deseen. Las

asignaciones múltiples sólo se podrán hacer entre un comando */SET* y un comando */END*. Los comandos erróneos que se inserten serán ignorados. Así mismo, no podrán haber otros comandos en medio de una lista o entre un comando */SET* y un comando */END*.

Si queremos inicializar el sistema de listas debemos usar los siguientes comandos:

```
/END;
/CLEAR;
```

Con estos dos comandos aseguramos que el sistema ha salido de cualquier definición de lista o asignación múltiple en la que esté insertada y borramos todo el sistema de listas.

5.4.3 Comandos generales del LIV.

En este apartado se van a comentar otros comando que no pueden estar agrupados en los apartados anteriores. Estos comandos tienen utilidades diversas, tales como interrogar por variables, refrescado de ventanas y parado del intercambio de variables.

/GET variable;

Este comando se utilizará cada vez, que desde el exterior a SiMon se desee conocer el valor de una variable. Tras enviar este comando, donde *variable* ha de ser un nombre válido de variable, el sistema responderá con una *asignación simple*, indicando el valor actual de la variable pedida.

/WHAT;

Interroga al sistema sobre las listas de variables que estén declaradas. Tras esto, el sistema, responderá devolviendo la identificación de todas las lista de variables que reconozca.

/WHATIS n;

Interroga al sistema acerca de una lista de variables concreta. Si el sistema la conoce, devolverá todos los nombres que están contenidos en esa lista.

/PAUSE;

Para la comunicación del sistema. Cuando el sistema recibe el comando */PAUSE*, cesa de enviar datos a través del canal de comunicación. Esto puede frenar el proceso que esté ejecutándose, pues si tiene que comunicar datos, y el canal está bloqueado, puede entrar en un bucle hasta que el sistema remoto libere la comunicación. Esta será liberada al enviarse otro mensaje.

/REFRESH win;

Refresca la ventana gráfica que se indique mediante el parámetro *win*. Este nombre no es otro, que el nombre asociado al comando *SHEET* que generó esa ventana gráfica.

5.5 Genesis.

En el capítulo 3 se explicó el papel que juega GeneSis dentro del entorno EASAP, además de hacer una presentación sobre cómo es su implementación. Aquí vamos a explicar más en detalle aspectos importantes del proceso de análisis léxico de una descripción LDAP, y de la forma de trabajo con las distintas opciones que nos permite la herramienta.

5.5.1 El analizador Lexer.

En el apartado 5.2 vimos la forma en que el usuario puede describir un circuito basándose en macrofunciones del lenguaje C. Como fue comentado en su momento, en este tipo de descripciones podemos usar todas las estructuras que se definen en el lenguaje C para realizar operaciones iterativas, creación de variables y demás operaciones propias del lenguaje C.

Mediante el uso de la herramienta ***lex*** definimos el lenguaje LDAP interpretado, creando nuestras propias palabras reservadas (comandos) y expresiones del mismo (sentencias). Simplemente no tenemos más que añadirle o indicarle qué reglas debe seguir para analizar el fichero fuente, el cual lo tomará a través de su entrada estándar. Se tratará en realidad de un fichero de texto donde con un formato adecuado se realizará exactamente el mismo tipo de descripción que se realizaba previamente con un fichero C. Sin embargo, en esta ocasión se tratará de un lenguaje propio, desarrollado específicamente, y cuyo análisis y posterior ejecución tiene por consecuencia llamar a las mismas funciones que anteriormente se usaban de forma directa por parte del usuario en el fichero fuente para el lenguaje C.

En realidad lo que estamos haciendo no es más que añadir por encima del nivel constituido por las macrofunciones que habíamos creado en el lenguaje C, una serie de facilidades para que el usuario desde un lenguaje de más alto nivel pueda describir el sistema sin necesidad de bajar al punto de conocer la sintaxis de un lenguaje de programación.

El único punto de entrada a esta librería, será la función denominada ***lexer()***. Esta función será llamada por el usuario para acceder a todas las facilidades del analizador léxico generado. A esta función sólo se necesita pasarle un único parámetro, es decir, el nombre raíz del sistema. Al hablar de nombre raíz indicamos que los ficheros que el ***lexer()*** usará, tanto de entrada como en su generación de datos de salida, todos tendrán el mismo nombre base, ya que simplemente se cambiará su extensión. Esta extensión será fija para cada uno de los ficheros a definir, salvo que la función ***lexer()*** esté siendo usada desde el entorno **GeneSis**, el cual sí permite alterar las extensiones por defecto que se usarán. Los ficheros que el ***lexer()*** puede proporcionar a su salida a petición expresa del usuario en el momento de su llamada, son los siguientes:

* un fichero binario donde se almacenarán todos los datos que mantienen las estructuras de memoria que describen el sistema,
* un fichero donde se generará un informe en texto estándar ASCII donde se recoge una descripción de todos los datos del circuito,
* un fichero con el formato LDV para el programa de monitorización **SiMon**, y
* un fichero fuente con la descripción del sistema a través del lenguaje LDAP compilado.

Función principal *lexer()*.

En primer lugar debe tenerse en cuenta que **lex** sólo realiza un análisis léxico del fichero LDV asignado. Si queremos saber si los comandos de este fichero están siendo usados en el orden correcto, habrá de implementarse los mecanismos necesarios para averiguarlo, ya que el analizador léxico que se puede generar con **lex** no nos proporciona esa información. Por tanto, ese reconocimiento deberá ser llevado a cabo mediante el software desarrollado en cada una de las funciones internas encargadas de realizar el análisis de cada una de las líneas del fichero fuente que el **lex** recibe a través de su entrada estándar.

De acuerdo con lo visto en capítulos anteriores, la descripción que un usuario hace de un sistema pasa por describir cada uno de sus componentes, así como por describir su composición interna: señales, pines, e incluso para aquellos elementos tipo array, bloque, o sheet, describir su composición a base de otros elementos como células, arrays o bloques. De acuerdo con esto, y de acuerdo con la forma en que se han diseñado las reglas del analizador léxico, cada línea del lenguaje será interpretada independientemente de las demás. Así en el proceso de análisis de éstas, podemos encontrarnos con dos grupos de errores:

* por un lado, los errores que implican la definición de un elemento (*CELL, ARRAY o BLOCK*) fuera de lugar, y
* por otro lado, los errores que implican la definición de un componente interno (*INOUT, FUNCTION, ...*) fuera de lugar.

La función ***lexer()***, genera a partir de su único parámetro (el nombre raíz del sistema) los nombres de los posibles ficheros de salida que el usuario haya escogido crear. Opcionalmente, éste puede escoger entre cuatro ficheros de salida, como ya se comentó anteriormente. Cada uno de estos ficheros tendrá una extensión asignada por defecto, pero que puede ser opcionalmente variada si está haciendo uso de la herramienta **GeneSis**, ya que éste le permite emplear un fichero de configuración.

Una vez que el usuario ha indicado qué ficheros de salida desea generar, la función ***lexer()*** llamará a ***crea_ficheros()***. Este función formará los nombres de los distintos ficheros de trabajo en función de las extensiones que correspondan. Por defecto el fichero fuente *LDAP* tendrá la extensión *".src"*.

A continuación la función ***lexer()*** acude a la función del sistema estándar del lenguaje C denominada ***freopen()***. Esta función, reasigna el fichero de trabajo a la entrada estándar (***stdin***). El efecto que se consigue con esta operación equivale a que en la línea de comando del sistema se use el operador '<' para asignar un fichero particular como entrada estándar. A partir de este momento la función ***lexer()*** comienza el análisis del fichero. Este análisis será dividido en dos lecturas completas del fichero:

* la primera lectura se limita a analizar léxicamente el contenido del fichero, pero sin ejecutar ninguna de las funciones de generación de estructuras de memoria.

* en la segunda lectura se realiza la ejecución efectiva de las funciones y de las órdenes contenidas en el fichero LDAP.

La forma que las funciones tienen de identificar en qué pasada nos encontramos, será mediante una variable global denominada ***analisis***, interpretada como sigue:

* Cuando adopta el valor 1, le indicará a las funciones internas que se está realizando la fase de análisis léxico. Por lo tanto, estas funciones realizarán todas las funciones implicadas en su ejecución, excepto la llamada a la función de librería que lleve a cabo de forma efectiva la acción deseada por el usuario.

* Cuando adopte el valor 0, las funciones internas harán todas las acciones implicadas en el análisis y, además, ejecutarán las funciones de librería que sean necesarias.

Para poder arrancar las operaciones del analizador léxico, se llama a la función ***yylex()*** que viene definida en las librerías de la utilidad **lex**. Su cometido es, simplemente, tomar los datos de la entrada estándar y agruparlos de la forma más apropiada de forma que cumplan las reglas definidas en el analizador léxico.

Función auxiliar *yywrap()*.

Una vez que la función ***yylex()*** llega al final de este fichero, llama a la función ***yywrap()***, que si bien viene igualmente aportada por la librería de la herramienta **lex**, se permite crearla de nuevo para modificar las acciones que realiza. La utilidad de esta función radica en que permite incluir en ella todas aquellas acciones que deseamos que el analizador léxico lleve a cabo al finalizar el análisis del fichero, y antes de devolver el control a la función que llamó a ***yylex()***. La forma en que interactúan las definiciones de ***yywrap()*** dadas por el software desarrollado en la librería ***lexer.lex*** y en la librería propia de la utilidad **lex** es muy sencilla, ya que si el software que se desarrolla no incluye su propia definición para esta función, el analizador léxico que se genere hará uso de la que se da en la librería del **lex** y, por el contrario, si se define una nueva función ***yywrap()***, ésta será usada en perjuicio de aquélla.

Sin embargo, debe tenerse en cuenta como interactúa a su vez la función ***yywrap()*** (desarrollada o de librería) con la función ***yylex()*** (recuérdese que ésta llamará automáticamente a aquélla cuando acabe de leer los datos del fichero de entrada). En particular, el valor que reciba al retorno de la ejecución de ***yywrap()*** será especialmente tenido en cuenta. Esta función puede devolver los valores 0 ó 1. La función ***yywrap()*** que aporta la librería del **lex** se limita, de acuerdo a la expresado en esta tabla, a devolver el valor 1, indicando al lex que el análisis léxico del circuito, y por tanto su trabajo, ha terminado. En cambio, la función ***yywrap()*** que se ha desarrollado como parte de la librería ***lexer.lex*** (y que sustituirá a la otra durante la fase de *"lincado"*) realiza un conjunto más complejo de operaciones, todas ellas encaminadas a nuestras necesidades en esta herramienta.

Sin embargo, en el software se ha incluido la definición de la función ***yywrap()*** para permitirnos la facilidad de realizar las dos pasadas léxicas que hemos comentado. De esta forma una vez que el lex ha terminado la primera pasada con la variable análisis a valor 1, llama automáticamente a la función ***yywrap()***.

Esta función comprobará el valor de la variable análisis; si está a 1 deducirá que efectivamente lo que se ha terminado es la pasada de análisis, y por lo tanto debe haber una segunda pasada. Su objeto entonces será reinicializar las variables globales, que probablemente han sido cambiadas en la primera pasada de análisis, reabrir de nuevo la entrada estándar de forma que el puntero de fichero vuelva al comienzo de éste, y retornar el valor 0.

Cuando la función ***yylex()***, que es quien llama a la función ***yywrap()***, detecta que ésta

devuelve un valor 0, entiende que aún no se ha terminado el análisis léxico, y vuelve a acudir a la entrada estándar a la búsqueda de más datos. Dado que en la función ***yywrap()*** nosotros previamente hemos devuelto el puntero del fichero al comienzo de éste, en definitiva lo que estamos haciendo es volver a analizar todo el fichero desde el principio.

Por supuesto, la función ***yywrap()*** sólo permitirá que se lleve a cabo la segunda pasada siempre que en la primera no se hayan detectado errores léxicos. Esto se lleva a cabo mediante una variable global denominada error que será puesta a valor 1 por cualquier función interna de este fichero que hemos generado, siempre que se detecte un error de algún tipo en el análisis que se está realizando.

Estos errores, como hemos mencionado previamente, pueden venir provocados por el hecho de que el usuario haya hecho uso de palabras reservadas fuera de lugar, tales como por ejemplo intentar definir un array dentro de una célula, o bien cuando intenta realizar la descripción de un componente interno de un elemento con un formato que no es el adecuado. Por tanto, en casos como este y en otros tantos que se verán a medida que se vayan describiendo las funciones internas, la variable error será puesta a valor 1.

En este caso inmediatamente la función ***yywrap()*** cuando toma el control al finalizar la primera pasada, detectará este valor, y se limitará a retornar con el valor 1 sin reabrir la entrada estándar. Por tanto, la función ***yylex()*** dará por terminado el análisis del circuito y mostrará el mensaje: *"Ejecución abortada por errores en el análisis léxico"*.

Finalmente la función ***yywrap()*** cuando efectivamente detecta que no ha habido errores y que por lo tanto se puede pasar a realizar la segunda pasada del análisis léxico, que como hemos comentado previamente lleva a cabo lo que es la ejecución propiamente dicha de las funciones asociadas a las palabras reservadas del lenguaje interpretado, como primera tarea llama a la función ***Init_System()***.

Ficheros de salida.

Como se comentó anteriormente, tenemos cuatro posibles ficheros. Estos ficheros recogen distintos formatos de presentación de la información contenida en memoria. En la **Tabla V.5** se muestran aquellos ficheros susceptibles de ser generados.

***Tabla V.5**: Ficheros generados por el **lexer** a petición del usuario*

FICHERO	CONTENIDO
Binario	Copia de las estructuras de datos de memoria.
Informe	Información ASCII sobre la arquitectura de trabajo.
LDV	Descripción LDV de la arquitectura.
Fuente C	Descripción de la arquitectura de trabajo mediante funciones C del LDAP compilado.

* En primer lugar, tenemos los ficheros binarios con los datos de las estructuras en memoria. La generación de este fichero se llevará a cabo a través de la función ***Save_System()*** de la librería desarrollada ***disco.c***.

* Para la generación de los ficheros de informe se usa la función ***Imprime_Memoria()*** descrita en la librería ***disco.c***. En el fichero ASCII de salida se encontrará información sobre todos los elementos que componen la arquitectura de trabajo.

* El tercer tipo de fichero de salida es la descripción LDV para el programa de monitorización **SiMon**. La tarea de generación de este ficehro la lleva a cabo la función ***genldv()***. Esta función, al igual que las demás, sólo precisa un puntero a la cadena de caracteres que contiene el nombre del fichero.

* El último fichero de salida puede resultarle bastante útil al usuario. El analizador léxico que se ha creado, proporciona, al tiempo que se realiza el análisis léxico del fichero de entrada, la posibilidad de traducir éste a su forma como fichero fuente C en el, como es lógico, se hará uso de la versión para compilación del lenguaje LDAP. Téngase en cuenta que lo que estamos generando no es un fichero ejecutable, sino que se trata del fichero fuente, es decir, un fichero texto con lenguaje C que habrá de ser compilado de igual forma que si el usuario lo hubiese escrito a mano.

La utilidad de esta última opción para generación de fichero de salida que proporciona el analizador ***lexer*** es manifiesta, ya que ello nos permite realizar una descripción rápida del sistema mediante el lenguaje LDAP interpretado (siempre será más rápido describirlo y generarlo mediante este lenguaje que hacer directamente el fichero fuente en C y realizar luego su compilación) y, a partir de este fichero generar una descripción mediante funciones C.

Esta utilidad se ha desarrollado para permitir la creación rápida de programas generadores de arquitecturas. El usuario podrá editar este fichero para incorporarle todas aquellas funciones que le permitan realizar una comunicación con el usuario durante la ejecución de éste. De esta forma, se podrá pedirle al usuario que indique el tamaño terminado de un array, qué tipo de células desea que contenga, cuáles serán las posiciones de un array con respecto a otro, o simplemente, se le preguntará qué tipo de arquitectura se desea generar, encargándose el programa de realizar todos los cálculos oportunos.

5.5.2 Definición de reglas del analizador lexer.

En principio, para cada comando del lenguaje interpretado habría que definir una regla que permita al analizador identificar el comando. Sin embargo, el uso del **lex** aconseja que el número de reglas independientes sea lo más pequeño que se pueda. Por tanto, en la medida de lo posible, las reglas se han de definir de forma genérica, dejando al software asociado a cada una discernir qué regla final es la que se está aplicando. De esta forma se obtiene un código más compacto y de menor tamaño para el analizador léxico generado.

Sin embargo, dado que ahora se trata de que el software subyacente realice parte del trabajo de reconocimiento, tampoco debemos ir al extremo opuesto y entregar reglas tan genéricas que sea el software quien lleve todo el peso del análisis, ya que en ese caso no se obtendría ningún beneficio del uso de la herramienta **lex**. Teniendo esto en cuenta se crearon las reglas que serán explicadas en el apartado dedicado a cada uno de los comandos del intérprete.

Aquellas expresiones que se verán como '{...}' (encerradas entre llaves) son definiciones de otras reglas a expandir. Funcionalmente, se asemejan a las sentencias ***#define*** del pre-

Tabla V.6: Definiciones a expandir en las reglas del analizador

NOMBRE	EXPANSION
N	[A-Za-z0-9_(),]+
S	[\n\t]
TS	(I|O|B|G)
TV	(BOOLEAN|BIT|CHAR|FLOAT|DOUBLE|INT8|INT16|INT32|UINT8|UINT16|UINT32)
MV	(V_CHDL|V_WAVES|V_LAYOUT|V_VERILOG|V_HILO|V_VHDL|V_SHEET|V_PINEXT|V_PININT)
MS	(CHDL|WAVES|LAYOUT|VERILOG|HILO|VHDL|SIMON)
G	(N|S|E|W|NE|NW|SE|SW)
R	"["{S}*[0-9]+{S}*:{S}*[0-9]+{S}*"]"
AR	"["{S}*[0-9]+{S}*(:{S}*[0-9]+{S}*)?"]"
NS	{N}(".'{N})*
RC1	"["{S}*[0-9]+{S}*([:*]{S}*[0-9]+{S}*)?"]"
RC2	"["{S}*[0-9]+{S}*""{S}*[0-9]+{S}*"]"
DF	"["{S}*[0-9]+{S}*"]"
RA	({RC2}{DF})|({DF}{RC2})
P	"["{S}*[0-9]+{S}*","{S}*[0-9]+{S}*"]"
DES	'[^'\n]*'
NF	'[A-Za-z0-9_.%#]+'
CE	(CELL|ARRAY|BLOCK|SHEET)
CI	(INOUT|INTERNAL|PLACE|CONNECT|MODEL)
LEV	(L_BODYCEL|L_CONTCEL|L_PINCEL|L_CONWIRE|L_CONBUS|L_TXTCEL|L_TXTARR|L_TXTBLO|L_TXTDES)

procesador del compilador C. El objeto de usar estas definiciones, que se declaran previas a las reglas del analizador, no es más que el de la comodidad en la lectura de las reglas definidas, dado que son expresiones normalmente largas y que se repiten con frecuencia. En la **Tabla V.6** se muestra la declaración de las reglas a expandir en las anteriores.

Definición de elemento.

Como se ha comentado previamente, existen cuatro tipos posibles de elementos que se pueden definir en nuestro lenguaje. Para cada uno de estos habrá un comando asociado del lenguaje *LDAP* interpretado y que se muestra en la **Tabla V.7**.

Tabla V.7: Comandos para la definición de elementos básicos

COMANDO	USO
CELL	Definición de célula
ARRAY	Definición de array
BLOCK	Definición de bloque
SHEET	Definición de sheet

El formato de todos estos comandos es el mismo, con la única excepción del propio nombre del comando, por lo que se adopta una única regla:

{CE}{S}+{N}{S}*"{"

donde:

{S} Representa un espacio en blanco, tabulador o retorno de carro.
{N} Nombre del tipo de elemento.
{CE} Comando básico.

Según puede observarse en la regla, tras el comando nos encontramos con {S}+, es decir, el comando debe ir seguido por, al menos, un separador (espacio en blanco, tabulador o retorno de carro). Sin embargo, tras el nombre del elemento que se va a definir, y antes de '{', puede haber o no separación y tal posibilidad de no existencia de separación se implementa con {S}*, donde el operador '*' indica que lo anterior se repetirá 0, 1, o más veces ('+' indica 'al menos una vez').

Con la definición a expandir de {CE}, que como se ve en la **Tabla V.6** representa a los cuatro comandos posibles para definición de elemento (ver **Tabla V.7**), se consigue reducir a una única regla lo que hubiese sido necesario presentar como cuatro.

Definición de las características internas de un elemento.

Una vez que se ha iniciado la definición de un elemento, se usarán los comandos adecuados para realizar la definición de su características internas (componentes internos de los elementos en definición). Los componentes a reconocer se muestran en la **Tabla V.8.**

***Tabla V.8**: Comandos internos.*

COMANDO	USO
INOUT	Definición de pines de E/S
INTERNAL	Definición de pines internos
PLACE	Posicionamiento de elemento
R, L o H	Idem, pero sólo en arrays
CONNECT	Conexión de señales
MODEL	Definición de modelos

Todos estos comandos (salvo los denominados R, L y H, usados para posicionar células en un array) tienen una sintaxis común. Por ello, con objeto de que el analizador léxico generado haga uso de un código ejecutable más compacto, se reunieron las reglas necesarias para estos cinco comandos en una única regla, la cual se muestra a continuación:

{CI}{S}*"{"

donde:

{S} Carácter separador (blanco, tabulador o retorno de carro).
{CI} Comando de definición de componente interno.

El carácter '*' indica que la '{' puede ir pegada al comando sin que haya separación alguna. La función interna a la que llama el analizador léxico al encontrar esta regla, ***elem_int()***, se encarga de distinguir el comando del que se trata y de bifurcar la ejecución del analizador hacia la función adecuada.

Definición de pin normal de célula.

Por pin normal de célula entendemos aquel que no sea interno a la misma, sino que tenga asociado una vía de E/S de datos hacia y/o desde la célula. Realmente, la definición de cada pin es independiente de la de los demás, si bien pueden estar todas agrupadas dentro del mismo comando INOUT (ver {CI} en el apartado anterior). Una vez que este comando ha sido reconocido, la definición de cada pin se captura no porque vaya precedida de una palabra especial reservada, sino que por la sintaxis con que se expresa y que la diferencia de todas las demás definiciones del intérprete.

Por tanto, la definición de cada pin normal de la célula no se trata del comienzo de ningún comando en particular, sino de la línea con la que se define ya específicamente dicho pin normal de la célula. Recuérdese que la definición de un pin precisa de diversos parámetros tales como:

* nombre del pin,
* flujo,
* tipo, y
* lado de conexión.

Estos son los parámetros que deben darse para la definición del pin, sin embargo, al usuario se le ofrece la posibilidad de que defina algunos parámetros por defecto, es decir, que no necesite especificarlos. En particular, será la definición del tipo de variable aquella parte de la regla que el usuario puede omitir. Caso de que esto sea así, es decir, que se omita el tipo de variable, se adoptará por defecto un tipo BIT (definido en ***basico.h*** como T_BIT).

Además, opcionalmente, caso de ser un pin con varios hilos (un bus), se puede especificar el rango de índices con que se distinguirán sus líneas. Por supuesto, todas las líneas de un bus tendrán asociado el mismo tipo de variable. Teniendo esto en cuenta, podemos encontrarnos cuatro formas diferentes de definir un pin, a las que llamaremos:

* Señal corta (un hilo sin especificar su tipo de variable).
* Señal larga (un hilo y especificando su tipo de variable).
* Bus corto (más de un hilo sin especificar tipo de variable).
* Bus largo (más de un hilo y especificando tipo de variable).

En principio para cada una estas haría falta una regla para el analizador léxico, que reconociera las diferencias de formato entre estas cuatro alternativas, ya que no es lo mismo presentar el nombre de una señal corta que el de un bus largo (éste incluiría, además del nombre del bus, el rango de líneas que lo forman y el tipo de variable). Sin embargo, entre las sintaxis que siguen la definición de señal y de bus hay ciertas diferencias tan marcadas que, si se dejase su distinción al software subyacente, habría que realizar un algoritmo excesivamente complejo. Por ello, finalmente, se decidió reducir las cuatro reglas iniciales a dos, siendo una de ellas para señales y, la otra, para buses. De esta forma se obtuvieron las reglas:

{N}({R})?{S}*:{S}*{TS}{G}{S}*(:{S}*{TV}{S}*)?[>;}/]

donde:

- **{N}** Nombre del pin.
- **{S}** Carácter separador.
- **{TS}** Tipo de flujo (entrada, salida,...).
- **{G}** Lado de conexión.
- **{TV}** Tipo de variable (BIT, BOOLEAN,...).
- **{R}** Rango de hilos que forman el pin.

Como puede verse, se hace uso del operador '?' para indicar la presencia de una parte opcional en la regla, es decir, que lo que hay inmediatamente a su izquierda puede estar presente o no. Sin embargo, no debe pensarse que podría haberse hecho uso del operador '*' (recuérdese que éste indicará que se puede repetir 0, 1 o más veces), pues cuando el uso del operador '?' indica que algo puede estar o no, pero si está, sólo lo estará 1 vez. No obstante, al usar '*' se indica que algo puede no estar o estar presente de forma repetida cuantas veces se quiera, lo cual no es lo que nosotros deseamos expresar. Así, p.ej., con el trozo de regla: {TV}?, queremos indicar que la especificación del tipo de variable pueda estar presente o no, pero que, de estarlo, sólo aparecerá una vez.

Mención aparte merece la sección final que indica qué carácter se permite como delimitador de la regla para definir un pin normal. Para ello, como puede observarse, se usa el operador de clase [], mediante el que se especifica que ha de escogerse uno de los caracteres de su interior. Así, al incluir en la regla la expresión: [>;}/], indicamos que uno cualquiera de estos caracteres se usará para delimitar el pin de la línea de definición de éste. Por supuesto, estos caracteres no se han escogido de forma aleatoria sino que siguen unas normas claras. Si bien su interpretación corre a cargo del software subyacente, que será explicado con detalle en próximos apartados de este capítulo, veremos ahora unas mínimas indicaciones acerca del empleo de cada uno:

Línea acabada con ";"

Esta será la situación más normal e indicará simplemente el final de la definición del pin sin ninguna acción adicional a realizar.

Línea acabada con "/"

Como se comentará posteriormente, en la descripción del sistema se permite el empleo de comentarios al estilo del C, señalados por /*...*/. Si este comentario se sitúa al final de la misma línea de la definición del pin, las normas sintácticas del lenguaje LDAP interpretado permiten que el ";" que normalmente marca el final de la definición se suprima, tomando como carácter delimitador la "/" del comienzo del comentario. En la explicación de las funciones de esta librería se verá con más detalle la forma de tratar esta situación.

Línea acabada con ">"

Este carácter es, en realidad, un operador del lenguaje LDAP interpretado, y cuyo empleo indica que el pin cuya definición delimita se ha de autoconectar con el pin que se defina a continuación.

Línea acabada con "}"

Dado que dentro del comando INOUT puede haber tantas definiciones de pin como se desee, en la última de ellas se puede suprimir el ";" delimitador de final de definición de pin y usar como delimitador el carácter "}" que cierra el ámbito de aplicación del comando INOUT.

Definición de pin interno.

La sintaxis para los pines internos se diferencia de los externos en que, por razones evidentes, no se especifican ni el flujo del pin (pues es interno) ni el lado en que se conecta (pues no se conecta a ninguno). Similarmente a como ocurría con la definición de los pines externos, para los internos también es opcional la especificación del tipo de variable, con lo que también podemos encontrarnos con dos sintaxis diferentes para realizar la definición de un pin interno:

* Corta: si sólo se indica el nombre del pin interno.
* Larga: si se indica el nombre y el tipo de variable.

Mediante el empleo del operador '?' (opción) podemos construir una única regla para ambos casos, dejando al software subyacente el reconocimiento de en qué caso nos encontramos. La regla queda como se muestra a continuación:

{N}({R})?({S}*:{TV}{S}*)?{S}*[;}/]

donde:

{N}	Nombre de pin.
{R}	Rango de hilos del pin.
{S}	Carácter separador.
{TV}	Tipo de variable.

Según se observa, esta regla también admite los mismos caracteres delimitadores de final que para los pines normales, a excepción del operador '>' de autoconexión, ya que los pines internos no se autoconectan con nadie (no en vano son internos).

Definición de células de un array.

Una vez que se ha iniciado la definición de una array con el comando ARRAY, el usuario señala las células que ocupan posiciones dentro del mismo mediante la indicación de un rango de filas y de columnas, siendo la intersección entre los elementos que pertenecen a cada rango la que nos da aquellos donde se situarán las células del tipo indicado.

Las expresiones que use el diseñador para realizar la definición con este formato deben comenzar con el carácter "R", "L" o "H". De esta forma se reconocerá que, a continuación, se indican aquellos elementos del array que se ocuparán con la célula indicada. Así, la regla que usa el analizador léxico se define como:

[LRH]{AR}{AR}{S}*={S}*{N}{S}*[;}/]

donde:

{N} Nombre del tipo de célula a poner.
{S} Carácter separador.
{AR} Rango de variación de índices.

Entre el carácter de inicio("R", "L" o "H") y los rangos de variación de los índices, y entre estos mismos, no puede haber ninguna separación (no hay carácter separador indicado por {S}), aunque internamente a cada rango sus límites sí pueden tener los caracteres de separación que se deseen (regla {S}*).

Conexión de señales de un sheet.

Durante el proceso de descripción de un sheet por parte del diseñador, se puede recurrir a dos formas de indicar la conexión entre señales de los diferentes elementos del sheet:

* indicando dos señales simples que se conectan (señal=señal), e
* indicando de una sola vez un grupo de señales a conectar.

En cualquiera de ambos casos, al final siempre se realizarán las conexiones de las señales una a una. Ambos formatos de las sintaxis para conexiones poseen las suficientes diferencias entre ellos como para no ser eficiente el uso de una única regla para ambos y dejar al software subyacente la tarea de distinguir cual de los dos formatos se está usando. Por ello, se optó por dejar dos reglas independientes, una para cada tipo de sintaxis, siendo la usada para señales simples:

{NS}{S}*={S}*{NS}{S}*(={S}*{N}{S}*)?[;}/]

y, la destinada a la conexión de señales de arrays:

{NS}{RC1}{RC1}"."[A-Za-z0-9_]+{S}*={S}*{NS}({RA})?"."[A-Za-z0-9_]+{S}*(={S}*{N}{S}*)?[;}/]

donde:

{N} Nombre de elemento.
{S} Carácter separador.
{RC1} Rango de variación de índices.
{RA} Rango de variación de índices con sintaxis alternativa.

La sintaxis para conexión entre grupos de señal es específica para su uso entre señales pertenecientes a células que forman parte de elementos tipo array. Jugando con los rangos de variación que se especifican en la línea, se puede lograr prácticamente cualquier combinación de conexionado entre las señales que forman parte de estos. Por ej., la conexión de una señal perteneciente a las células que ocupan una columna de un array se puede realizar con los siguientes elementos:

* una señal simple, es decir, todas las señales conectadas a la misma, o
* un grupo de señales del mismo o de otro array, quedando las señales conectadas una a una.

Los nombres que se indican para los señales son nombres completos, es decir, toda la jerarquía de posicionamiento de la señal (nombres de elementos separados por un "." y comenzando desde el elemento situado directamente sobre el sheet hasta llegar a aquella

célula que contiene a la señal). Cada componente del nombre completo es de la forma {N}, la cual admite como caracteres formantes de un nombre a algunos que no son válidos para nombrar un pin (o señal). Así, para especificar el nombre del pin, debemos añadir un elemento extra tras {N}, para especificar que no se admitirán como nombre de éste aquellos caracteres tales como: '(', ')' y ','.

Posicionamiento de un elemento.

Hay una única forma para dar el formato con que se señala la operación de posicionamiento de un elemento dentro de otro, ya que, si bien se admite la posibilidad de especificación opcional de datos, mediante el empleo del operador '?' se permite usar una única regla. Esta se ha diseñado de la siguiente forma:

{N}{S}*={S}*{N}{S}*:{S}*{P}{S}*(:{S}*{P}{S}*)?:{S}*(-)?[0-9]+{S}*[;}/]

donde:

{N}	Nombre lógico a dar al elemento.
{S}	Carácter separador.
{P}	Valores numéricos de posicionamiento.

La especificación del segundo grupo de valores numéricos es opcional (regla '({P})?'). Además también se toma como opcional el signo del ángulo de rotación que se indica al final. Si bien éste está comprendido entre ±360°, esta comprobación no la realiza el analizador léxico, el cual se limita a encontrar un número (puede ser negativo o no) formado por una cantidad indeterminada de dígitos, dejando al software subyacente analizar la coherencia de los datos especificados en esta regla.

Definición de modelo de visualización.

La definición de modelos conlleva un pequeño comentario de descripción y, normalmente, el nombre del fichero donde se puede encontrar la definición del modelo. Todo ello irá siempre precedido del propio nombre de éste (VHDL, VERILOG, ...). El nombre de fichero es opcional, si bien debe ser indicado siempre que el usuario quiera destinar al visualizador **SiMon** la descripción del sistema. Teniendo en cuenta todos los parámetros que deben incluirse en la definición del simulador, la regla creada para el analizador léxico se concreta en la siguiente:

{MV}{S}*={S}*{DES}{S}*(:{S}*{NF}{S}*)?(:{S}*NULL{S}*)?[;}/]

donde:

{MV}	Nombre del modelo de visualización.
{DES}	Descripción del modelo.
{NF}	Nombre del fichero.

Las descripción que incluye el usuario debe ir encerrada entre comillas simples y puede incluir cualquier carácter ASCII, es decir, lo que haya entre las comillas simples que lo delimitan (no se ven aquí pues van incluidas en la definición {DES}) pasará por el analizador de forma transparente sin verse afectado ni analizado.

Como puede apreciarse en la definición del nombre de fichero, {NF}, no se permiten espacios

en blanco entre las comillas que delimitan el nombre y éste. Además, los caracteres que deben figurar en él habrían de estar entre los que el UNIX admite como integrantes aptos para un nombre de fichero. Sin embargo, no se realiza ninguna comprobación de si la construcción del nombre es correcta, tarea que se deja al software subyacente.

Comentarios incluidos.

Si entendemos por comentario, todo aquel trozo de texto cuyo contenido es ignorado a efectos del proceso de generación del sistema descrito por el diseñador, entonces podemos decir que existen dos tipos posibles de comentarios que el usuario puede incluir en su descripción del sistema. Por supuesto, como ya se ha dicho, dado que se trata de *comentarios*, el contenido de ambos será ignorado por el software a efectos de generación de datos, si bien cada tipo de comentario será tratado de forma diferente.

Por un lado, están aquellos comentarios que el usuario escribe en el fichero fuente como guía personal a la hora de leer y/o editar el contenido de este fichero. Funcionalmente, actúan igual que los comentarios que se ponen en el fuente de un lenguaje de programación (C, PASCAL, ...). Su contenido será simplemente ignorado tanto a efectos de análisis como de ejecución. Su sintaxis es similar a la de los comentarios en C, es decir:

/ ... cualquier texto ... */*

Por tanto, al analizador léxico se le incluyó una regla que le indicara que ignorase cualquier texto que estuviese delimitado de esta forma. Las normas que rigen la construcción de estos comentarios son las mismas que para el lenguaje C, es decir:

* No se permiten comentarios anidados.
* Dentro de un comentario puede haber cualquier combinación de caracteres a excepción de "/*" y "*/" (si aparecen ambos, estaríamos en el caso de comentario anidado).
* El contenido puede ocupar tantas líneas como se desee.

Teniendo en cuenta estas normas, la regla que se añadió al analizador léxico tiene la siguiente forma:

```
"/*""/"*([^*/]|[^*]"/"|"*"[^/])*"*/"
```

donde se recogen todas la normas de construcción anteriormente especificadas.

El segundo tipo de comentarios que se admiten en la descripción del sistema, también son ignorados por el analizador a efectos de análisis de su contenido pero, a diferencia del anterior tipo comentado, éste es enviado a la pantalla. Su utilidad reside en el hecho de permitir al usuario el mostrar, en la ventana de salida de texto, mensajes que le vayan indicando los puntos por los que va pasando el analizador léxico y, de esta forma, llevar un seguimiento de la evolución del proceso de generación del sistema a partir de la descripción dada por el usuario. Al igual que en el caso anterior, hay una serie de normas que deben cumplir los comentarios que se construyan de este tipo:

* Ocuparán una sola línea de texto.
* El comentario se marcará con un carácter "#" al comienzo de la línea.
* En su contenido se admite cualquier combinación de caracteres, incluyendo el propio "#", siempre que no se exceda de la línea en que se comenzó el comentario.

Si se razona a partir de estas normas, se llega a la conclusión de que el comentario, para que sea reconocido como tal, debe de estar sólo en la línea en que se encuentra y que no puede haber ningún carácter (perteneciente a él o no) a la izquierda del carácter "#". La regla que se define al analizador léxico para hacer esto es tan sencilla como:

```
^#.*
```

donde con el operador "^" se indica que "#" debe ser el primer carácter de la línea, y el operador "." hace referencia a "cualquier carácter excepto el retorno de carro". Recuérdese que el operador "*" indica que lo que hay a la izquierda se repite 0, 1 o más veces.

Final de comando.

En el lenguaje LDAP interpretado, el final del ámbito de aplicación de un comando se puede delimitar de varias formas:

* para aquellos que ocupan sólo una línea, su final se marca con ";" o "/", mientas que,
* para los que ocupan varias líneas, hay que especificar el final del comando con "}".

En el primer caso no hay necesidad de proporcionar una regla léxica adicional, ya que en la propia regla de definición de cada comando, se incluye el carácter delimitador del final. Sin embargo, para los comandos que se extienden a lo largo de varias líneas, en algunos casos se necesitará una regla adicional que recoja la "}" que cierra el comando.

Recuérdese que, según las normas del lenguaje LDAP interpretado, una línea interna de un comando debe terminar en ";" pero, si es la última, puede tomarse como delimitador de final el carácter '}' de cierre del propio comando. Si esto es así, no hay necesidad de regla adicional. En cambio, si esa última línea también es terminada por ";", la llave de cierre quedará fuera del patrón encontrado y no será recogida por el analizador como parte de la definición de este comando. Así, para evitar un error, habrá de ser recogida aparte con una regla propia adicional. Esta, como se ve a continuación, es la más sencilla de las que se le dan a nuestro analizador:

```
"}"
```

Con esto, se indica al analizador que siempre que encuentre un carácter '}' que no forme parte de algún patrón de regla anterior, se tome como marca de final de comando y sea llamada la función adecuada (***final()***).

Inclusión de ficheros.

En varias ocasiones anteriores, se ha hablado de que una de las salidas generadas por nuestra herramienta es la de un fichero con el formato LDV específico para el programa monitor **SiMon**. Este formato LDV admite la posibilidad de que el usuario indique ficheros que se desean incluir tal y como si hubiesen sido tecleados directamente dentro de él.

El lenguaje LDAP (tanto el intérprete como el compilado) contemplan la sintaxis necesaria para que el usuario señale qué ficheros deberán indicarse como incluidos en el momento de

generar el fichero LDV. Esta sintaxis se plasma en la regla dada al analizador léxico, que es la siguiente:

USE{S}+{N}{S}*[;/]

donde:

{S}	Carácter separador.
{N}	Nombre del fichero.

En esta ocasión se ha usado la definición {N} en lugar de {NF} pues esta última incluye las comillas simples y, para el caso actual, éstas no son necesarias. Además, como puede observarse por {S}+, debe haber al menos un carácter separador entre el nombre del comando y el del fichero.

Dibujo de elementos pasivos.

Por elementos pasivos se entienden todos aquellos elementos destinados únicamente a ser dibujados por el programa monitor **SiMon**. Por tanto, al igual que para el caso de los ficheros a señalar como incluidos en el formato LDV, también se da al diseñador la posibilidad de especificar directamente aquellos elementos pasivos a ser dibujados.

Dado el diferente número de parámetros que precisan cada uno de estos comandos, se ha decidido dividir el grupo de estos en tres reglas independientes, de forma que se permite un uso más eficiente de las capacidades del analizador léxico generado y una menor carga para el software subyacente. De acuerdo con esto, las reglas definidas son las siguientes:

* regla 1:

TEXT{S}*"{"{S}*{LEV}{S}+{DES}{S}+[0-9]*{S}+[0-9]*{S}+[0-9]*{S}*"}"

* regla 2:

(BOX|POL){S}*"{"{S}*{LEV}{S}+{LEV}({S}+[0-9]*)*{S}*"}"

* regla 3:

(REC|CIR|LIN){S}*"{"{S}*{LEV}({S}+[0-9]*)*{S}*"}"

donde:

{S}	Carácter separador.
{LEV}	Nivel de color escogido para el elemento pasivo.
{DES}	Texto a imprimir.

La primera de ellas recoge el comando para dibujar una línea de texto en la pantalla de monitorización de **SiMon**, en las coordenadas indicadas y con el nivel de color deseado.

La segunda de ellas permite el dibujo de cajas (BOX) y polígonos (POL), con los vértices situados en las coordenadas indicadas y los niveles de color adecuados (contorno y fondo). Un polígono, en principio, puede tener tantos vértices como desee indicar el usuario.

La tercera de las reglas permite el dibujo de diversos elementos tales como círculo (CIR), rectángulo (REC) y línea (LIN). La diferencia existente entre rectángulo y caja queda patente en la reglas léxicas dadas a cada uno. Básicamente, una caja es un rectángulo al que, además, se le ha coloreado el fondo.

Definición de autosímbolo.

Cuando el diseñador desea realizar una representación del sistema que ha descrito, normalmente generará una salida en fichero de formato LDV para el visualizador de monitorización **SiMon**. Los elementos que se presentan con símbolo propio serán todas aquellas células que se hayan situado en el sistema, o bien de forma independiente, o bien formando parte de un elemento mayor, tal como un array o un bloque.

La función ***genldv()*** responsable de la generación del formato LDV, realiza un cálculo automático de cómo debe ser el símbolo de la célula que se va a representar. Sin embargo, la herramienta proporciona al diseñador la posibilidad de sustituir éste por su propio símbolo, el cual especificará mediante el dibujo de los elementos pasivos adecuados.

Ahora bien, para evitar que sea generada la descripción automática de símbolo para cada elemento (célula), se proporciona al usuario un comando con el que pueda marcar qué elementos van a generar su propio símbolo y cuáles otros van a tener uno introducido por el propio usuario. La regla léxica para este comando se muestra a continuación:

AUTO{S}*"{"{S}*{N}(";"{S}*{N})*{S}*"}"

donde:

{S} Carácter separador.
{N} Nombre de tipo de elemento cuyo símbolo debe ser autogenerado.

Resto de información.

Para el caso de que se cometa algún error gramatical en el lenguaje y que el analizador léxico encuentre un patrón que no encaja con ninguno de los que le han sido definidos, debe incluirse una regla que recoja cualquier cosa que no coincida con ninguna de las reglas anteriores.

```
.       |
\n
```

El operador '|' indica que la acción a tomar por la regla asociada será la misma que para la siguiente regla que se especifique. Téngase en cuenta que, de la forma en que se ha puesto, en esta regla se tomará como coincidente cualquier carácter ASCII, incluidos todos los caracteres de espacio, tabulador y retorno de carro (además de cualquier otro carácter ASCII). Sin embargo, un carácter de este tipo no se puede considerar un error y el software subyacente deberá ignorarlo. No obstante, cualquier otro carácter que no sea un espacio, tabulador o retorno de carro, y que sea "capturado" por esta regla se puede considerar un error gramatical o, probablemente, el inicio de un error gramatical y, como tal, será denunciado por el analizador.

Interfase para generación de estructuras con LDAP compilado.

Como se ha comentado en anteriores ocasiones, el empleo del lenguaje LDAP compilado implica la creación, por parte del usuario, de un fichero fuente con la descripción de la arquitectura a generar. En este apartado se explicará cómo se genera el programa ejecutable.

La descripción de una arquitectura utilizando LDAP compilado, se realiza siempre dentro de la función ***Main_Sheet()***. Para poder generar el programa ejecutable hay que aportar una

función ***main()*** que marque el punto de entrada al programa ejecutable generado. Esta función la aporta la librería ***mainbas.c*** y contendrá las instrucciones adecuadas para llamar a la función principal ***Main_Sheet()*** que contendrá las funciones para la descripción de la arquitectura. Una vez que se retorna de esta función ya tendremos en memoria todas las estructuras con los datos de la arquitectura generada. A partir de este momento, las acciones se encaminan a realizar las salidas de datos a ficheros. Los diferentes ficheros que el usuario puede obtener de la ejecución del programa compilado son los siguientes:

* Fichero binario con una copia exacta de la información que mantienen las estructuras de memoria. Se genera con la función ***Save_System()*** (librería ***basico.c***).

* Fichero ASCII con la descripción general de cada uno de los elementos generados como parte del sistema descrito. Se genera con la función ***Imprime_Memoria()*** (librería ***disco.c***).

* Ficheros de simulación con formatos diversos según el tipo de simulador que vaya a utilizarse.

En los dos primeros casos, los nombres de los ficheros de salida se generan a partir del nombre del programa ejecutable que se generó en la compilación, añadiéndoles la extensión *.bin* y *.info*, para el fichero binario y el fichero de información ASCII, respectivamente. Sin embargo, dado que estos ficheros también se pueden generar a partir del LDAP interpretado, para que no haya solapamiento de información se evita ésta anteponiendo, a estos nombres, un carácter "_" (*underscore*). Así, por ejemplo, para un ejecutable de nombre '*sistema*' se obtendrían los ficheros: '*_sistema.bin*', '*_sistema.info*' y '*_sistema.ldv*'.

En cambio, para los ficheros destinados a simuladores, los nombres que tengan se obtendrán de la propia descripción del sistema, es decir, a través de los comandos ***Def_Simul()*** especificados por el usuario en el fichero fuente.

Interfase para generación de fichero LDV.

Las funciones necesarias para la generación de un fichero LIV se encuentran en la librería ***genldv.c***, la cual habrá de ser compilada con un fichero adecuado que aporte una función ***main()*** que marque el punto de entrada al ejecutable. Este fichero recibe el nombre de ***mainldv.c***.

La librería ***genldv.c*** será accedida siempre a través de una función principal llamada ***genldv()*** y que será la encargada de llamar al resto de las funciones internas de la librería. El único parámetro que precisa esta función principal es el nombre que se va a dar al fichero de salida. Al retornar de esta función, éste ya habrá sido creado y contendrá la descripción visual del sistema con el formato LDV.

La función ***genldv()*** asume que los datos que describen el sistema se hayan presentes en la memoria antes de ser llamada. Por tanto, como la ejecución de este programa es independiente de la del que generó las estructuras en memoria y, por tanto, aquéllas ya no se encuentran en ésta, hemos de regenerar el contenido de esas estructuras **antes** de llamar a la función ***genldv()***. Para ello, haremos uso de la llamada a la función ***Restore_System()***, de la librería ***disco.c***. Esta función lee los datos del fichero binario que mantiene la información del

sistema, cuyo nombre del fichero le es pasado como único parámetro.

Los nombres del fichero de entrada, con los datos binarios, y el fichero de salida, con el formato LDV, se generarán internamente a partir del nombre raíz del sistema. Este nombre raíz es solicitado desde la función ***main()*** aportada por la librería ***mainldv.c***, que se usó para compilar la ***genldv.c*** y obtener el ejecutable actual, y no debe incluir ninguna extensión (aunque, si la incluyera, se ignoraría). A partir del nombre introducido por el usuario cuando el programa lo solicitó se forma el nombre del fichero que contiene los datos binarios (extensión *.bin*) y el del fichero de salida para el visualizador (extensión *.ldv*).

Interfase con analizador léxico.

Además de los interfases descritos hasta este momento, el usuario tiene la posibilidad de situarse a más alto nivel, sin llegar todavía al empleo del entorno **GeneSis**, y hacer uso de todos ellos desde el interfase proporcionado por el analizador léxico, al que denominaremos ***lexer.***

Este hará uso de la librería de funciones desarrolladas como parte del analizador léxico generado con la herramienta ***lex*** del Sistema Operativo **UNIX**. Esta librería, ***lexer.lex***, contendrá todas aquellas funciones necesarias para el análisis de los comandos del lenguaje LDAP interpretado y su derivación en llamadas apropiadas a las librerías básicas comentadas en apartados y capítulos anteriores.

Todas las funciones desarrolladas en las demás librerías están accesibles a este interfase, ya que se compila con todas aquéllas. Además, su función ***main()*** (aportada por la librería ***mainlex.c***) permite al usuario la posibilidad de escoger qué ficheros de salida de los comentados en apartados anteriores quiere obtener, pudiendo incluso no escoger ninguno (proceder que no tendría mucho sentido ya que una vez generadas las estructuras en memoria, el programa finaliza y el contenido de aquéllas se pierde).

5.5.3 Funciones del panel principal de GeneSis.

En el apartado 3.8.1 dedicado al interfaz gráfico de GeneSis, se comentó cuáles eran las utilidades que disponía el usuario durante una sesión de trabajo, y que podían ser invocadas desde el panel de control. Ahora explicaremos algunos aspectos importantes de esas opciones.

5.5.3.1 Lexer: el analizador léxico.

La función asociada a esta opción del panel de comandos se denomina ***lexer_proc()***. A través de ésta, se realiza la llamada a la librería de funciones ***lexer.lex***. La entrada a esta librería se realiza mediante la función ***lexer()***. El parámetro que se le pasa a esta función lo constituye el nombre raíz del sistema, por tanto es importante verificar que realmente se ha definido éste. La función desarrollada ***hay_raiz()*** realizará esta comprobación y, en caso de no haber sido definido el nombre raíz del sistema, mostrará un panel de advertencia al usuario y abortará la ejecución de la función ***lexer_proc()***.

Además, debe ser realizada otra comprobación previa a la llamada a la función ***lexer()***. Debe

tenerse en cuenta que la ejecución de esta función tendrá por resultado la generación en la memoria de las estructuras que contendrán los datos del sistema descrito por el usuario. Por ello, y dado que en cada momento sólo puede haber un sistema en la memoria, es lógico que deba realizarse una verificación de que no haya datos presentes en la memoria en el momento de llamar a la función ***lexer()***. Esta verificación se lleva a cabo mediante la función ***hay_datos()***, que en caso de encontrar datos presentes en la memoria (valor del puntero ***sys*** distinto de NULL) mostrará un aviso de advertencia hacia el usuario, permitiéndole las tres opciones siguientes:

"Seguir, Liberar memoria"

Esta opción hará que se continúe adelante con la llamada a la función lexer, si bien previamente se liberará la memoria usada por los datos actualmente presentes, es decir, se destruirán los datos actualmente en memoria.

"Seguir, Salvar datos"

Hacer click con el ratón en el botón asociado a esta opción provocará que se continúe adelante con la llamada a la función lexer, si bien previamente se realizará una copia en disco de las estructuras de datos actualmente en memoria.

"Cancelar"

Cuando el usuario selecciona esta opción, se aborta el proceso de ejecución de la función lexer. Las estructuras de datos en memoria permanecen intactas, y sin ver alterado su contenido.

5.5.3.2 Restauración de datos de disco.

Además de la generación de las estructuras en memoria directamente a partir de la descripción dada por el usuario, mediante la función ***restore_proc()*** se puede regenerar las estructuras de datos a partir del contenido de un fichero binario de formato adecuado. Este fichero puede haber sido creado por el usuario durante la presente sesión o en alguna anterior.

Dado que el efecto de la función ***restore_proc()*** es el de regenerar las estructuras de memoria, y debido a las mismas razones que para la función ***lexer_proc()***, habrá que realizar una comprobación previa de si se ha definido nombre raíz para el sistema (función ***hay_raiz()***) y de si ya existen datos presentes en memoria (función ***hay_datos()***). En ambos casos, se seguirá la misma secuencia de acciones que las explicadas para el procedimiento ***lexer_proc()***.

Si todo va de acuerdo a lo deseado por el usuario, se llamará a la función ***Restore_System()***, desarrollada en la librería ***disco.c***. El único parámetro que precisa esta función es el nombre del fichero en el cual se encuentran los datos que se van a restaurar. Este nombre de fichero se genera a partir del nombre raíz del sistema, seguido de la extensión definida por el usuario para los ficheros de este tipo (campo ***bin*** de la estructura FILEXT, que podrá ser alterado por el usuario usando el botón de SETUP del panel general, submenú de configuración).

5.5.3.3 Copia en disco de los datos de memoria.

Una vez que el usuario ha llevado a cabo la generación de las estructuras de memoria por

cualquiera de los caminos que tiene a su disposición, es decir, tanto a través del analizador léxico aplicado sobre el lenguaje LDAP interpretado como a través de la ejecución del programa generado con el lenguaje LDAP compilado, tiene la opción de realizar un almacenamiento permanente en fichero de los datos que ha obtenido en la memoria.

Si este es el caso, acudirá a la función ***save_proc()*** a la cual accede a través de la pulsación con el ratón sobre el icono adecuado. Esta función, dado que precisa de la llamada a una función de librería para realizar el almacenamiento en fichero de los datos, necesita construir este nombre. Este nombre, como todos los demás nombres que se necesitan durante el programa (nombres de ficheros, por supuesto), se construye a partir del nombre raíz y añadiéndole el campo adecuado del fichero de la estructura ***filext***. En esta estructura se guarda, en cada uno de sus campos, la extensión adecuada para los posibles ficheros de salida que el usuario desee generar.

Por tanto, es imprescindible que el usuario haya fijado un nombre raíz para el sistema y, por lo tanto, es necesario una comprobación previa de que esto ha sido hecho así. Como en anteriores comandos, nos valemos para ello de la función desarrollada ***hay_raiz()***. Esta función, caso de verificar que no se haya definido ningún nombre raíz para el sistema en el momento de llamar a ***save_proc()***, mostrará al usuario un panel de advertencia e inmediatamente anulará el resto de la ejecución de esta función. Sin embargo, si realmente se ha definido un nombre raíz, continuará la ejecución construyendo el nombre del fichero en que va depositar los datos almacenados en memoria añadiéndole al nombre raíz, como ya se ha dicho, el campo bin de la estructura ***filext***. La función que realizará el almacenamiento en disco recibe el nombre de ***Save_System()***. Esta función acepta como único parámetro el nombre del fichero de salida en el cual descargará en formato binario el contenido de las estructuras actualmente en memoria y forma parte de la librería ***disco.c***.

5.5.3.4 Ejecución de programas compilados.

Dentro del interfaz con el usuario **GeneSis**, se tiene la opción de ejecutar el programa compilado, es decir, el programa ejecutable que se haya generado a partir de la compilación del fichero escrito por el usuario con el lenguaje LDAP compilado. Para ello, la ejecución de este programa se realiza a través de la ventana de comandos creada en el interfase (ventana TTY) a la cual se enviará un comando mediante una función apropiada que nos proporciona la librería del ***OpenWindows***.

Este comando, denominado ***ttysw_input()*** permite enviar a esta ventana una cadena de caracteres de forma que la ventana de comandos la trate como si hubiese sido introducida directamente a través del teclado.

Este programa, en su ejecución generará siempre tres ficheros de salida. Por un lado, un fichero binario que contiene una imagen exacta de las estructuras en memoria, de las estructuras que creó ese programa en particular en memoria. Por otro lado, un fichero informe donde se contiene información textual de la descripción que el usuario ha realizado, con parámetros informativos acerca de la cantidad de memoria absoluta y porcentual que ocupa cada uno de los elementos por él descrito. Finalmente, el fichero de formato LDV para el programa monitor **SiMon**.

Es importante no confundir entre las estructuras de memoria que ya podrían haber sido

creadas dentro del interfase **GeneSis** y las que serán creadas por la ejecución de este programa.

Por ejemplo, el comienzo de la sesión de trabajo usa el comando de ***restore_proc()*** en el cual, a partir de un fichero binario actualmente en disco, regenera las estructuras de datos en memoria. Inmediatamente después, supongamos que el usuario desea ejecutar el programa compilado. La ejecución de este programa generará unas estructuras de datos en memoria paralelas a las anteriores. Estas estructuras de datos no se solapan, ni se cruzan, ni se molestan mutuamente, sino que simplemente son dos estructuras paralelas y estando la segunda de ellas, es decir, la creada por el programa compilado, activa o realmente existente durante la ejecución de este programa.

Dado que al terminar la ejecución del programa las estructuras desaparecen, el programa preserva su contenido generando un fichero binario, así como el fichero informe y LDV comentados anteriormente. Una vez que la ejecución de este programa finaliza, las estructuras de datos por él creadas se liberan, pero las que habían sido creadas ya anteriormente a partir de la función ***restore_proc()*** permanecen intactas sin haberse visto afectadas por la ejecución de este programa compilado.

Para que los nombres de los ficheros generados por el programa compilado no coincidan con los nombres de los ficheros generados durante el proceso de análisis léxico, ésto comenzarán con un "underscore". Las extensiones que se usarán se encontrarán en el fichero de configuración ***genesis.setup***, en el cual el usuario habrá indicado mediante los comandos adecuados la extensión para cada uno de los tipos de fichero que desee generar, así como otra serie de parámetros que serán comentados en su momento.

5.5.3.5 Ejecución del programa monitor SiMon.

Para invocar a SiMon desde GeneSis podemos picar en el icono correspondiente del panel de control. Esta acción realiza una llamada a la función ***ttysw_input()***, que envía a la ventana TTY de GeneSis el comando correspondiente comenzando la ejecución de SiMon, el cual comenzará con su ejecución normal, y el usuario trabajará con él, exactamente como si lo hubiese llamado él directamente desde una ventana de comandos del sistema.

5.5.3.6 Visualización de la jerarquía del sistema.

Una utilidad interesante que se proporciona al usuario, consiste en permitirle examinar la jerarquía del sistema que él ha descrito y que ha sido generada. Esta función se realiza picando con el ratón en el icono adecuado, lo cual tendrá como consecuencia la llamada a la función ***jerarquía_proc()***, que previa a su ejecución, realiza dos filtrados de error:

* Primero, comprobar que realmente existe un sistema definido, es decir, un nombre raíz para el sistema.

* Segundo, verificar que en memoria existen estructuras de datos conteniendo información de un sistema, lo cual es evidente, dado que precisamente el objeto de esta función es analizar esas estructuras de datos, y generar una información ASCII

para el usuario en la cual se presenta todo el árbol jerárquico del sistema.

Si realmente existe una estructura de datos en memoria, la ejecución de función de jerarquía continúa normalmente. Pero si no es así, se mostrará un panel de advertencia hacia el usuario. Si no se produce ningún contratiempo, la ejecución de la función ***jerarquia_proc()*** continuará adelante.

Inmediatamente se procede a comenzar el análisis de toda la estructura de datos presente en memoria. Recuérdese que lo que se tiene en memoria son los datos de un sistema, y que un sistema puede estar formado por diferentes sheets. Mediante un bucle adecuado barremos todos estos, es decir, barremos la lista que nos encadena los sheets que el usuario ha descrito, y para cada uno de estos se acude a la función desarrollada ***lista_jerarquia()***.

Las estructuras de datos que se mantienen en memoria para la descripción de los sheets, están formadas por listas enlazadas, organizadas de forma jerárquica, es decir, son varios niveles de listas, donde a partir de un elemento particular de una de ellas puede arrancar otra paralela que consideramos un nivel inferior.

Por lo tanto, la función ***lista_jerarquia()***, para recorrer de forma eficiente los niveles jerárquicos de esta organización, ha sido desarrollada de forma recursiva, es decir, puede llamarse a sí misma si fuese necesario para analizar niveles jerárquicos inferiores al que se encuentra actualmente.

La jerarquía se presenta en el interior de un ventana tipo editor de texto y consiste en presentar cada nivel jerárquico en una columna. La información presentada se tratará de una secuencia de líneas, en las que cada nivel jerárquico supone una entrada de tabulador hacia la derecha. Por lo tanto, es evidente que en la primera columna de la izquierda tendremos el nombre de los sheets. Debajo de cada nombre y corrido dos tabuladores hacia la derecha, tendremos la lista de elementos colocados directamente sobre este sheet.

Cuando se llegue a un elemento que no es una célula, hemos de mostrar la jerarquía de los niveles que dependen de ese elemento. Es decir, por ejemplo, de un array o de un bloque habremos de mostrar la jerarquía de los elementos contenidos en él, para lo cual volvemos a llamar a la función ***lista_jerarquia()*** de forma recursiva.

Para cada uno de los elementos que se representan, la información que se indica es la siguiente:

* primero, el nombre lógico del elemento posicionado, y

* segundo, y entre corchetes, el nombre del tipo de elemento.

Dentro de la ventana de jerarquía, disponemos de un panel que proporciona al usuario dos botones sumamente útiles para el tratamiento de la jerarquía:

* un botón rotulado como "**Acerca de...**", el cual proporciona información contenida en la estructura de datos en memoria que guarda las características del elemento que se haya escogido.

* otro botón denominado "**Conectado a...**", el cual sólo afecta a los nombres lógicos de

- las célülas, es decir, el nombre que han recibido en el momento de posicionarse sobre el sheet. El empleo de este botón proporciona una lista de aquellos otros pines del sheet a los cuales se conectan cada uno de los pines de la célula afectada.

Si seleccionamos el nombre de un tipo de célula y se invoca "**Acerca de...**" se abriría un panel de información en el cual se muestra:

* el nombre de la célula,

* entre corchetes su tipo (es decir, entre corchetes iría la palabra "*Celula*"),

* una indicación del número de pines de la que consta, pormenorizando el número de pines de cada tipo (es decir, el número de pines de entrada, de salida, etc.),

* una indicación del número de modelos de visualización de que se dispone para esa célula, y

* si se trata de un tipo de célula con generación automática de símbolo.

Suponiendo que no se trate de un tipo de elemento, sino del nombre de un elemento posicionado en un sheet, entonces se pasará a realizar una búsqueda a lo largo de esta lista encadenada es tan sencilla como:

* Primero, leer el valor actual del índice en el que se encuentra el punto de inserción dentro de esta ventana.

* Con este valor comenzar a recorrer toda la lista encadenada hasta encontrar un índice que sea mayor (la marca guardada en la lista se convierte a índice antes de cada comprobación).

Por otro lado, la función asociada a la solicitud del usuario de información de conectividad (botón "**Conectado a...**") nos da la siguiente información:

* nombre completo del elemento que el usuario ha seleccionado,

* lista de todos los pines que la constituyen (a excepción de los pines internos, dado que éstos no se conectan con nadie exteriormente) y, para cada uno de estos pines, una lista de todos los demás pines del sheet que se conectan a este pin.

5.5.3.7 Generación de ficheros con formato LDV.

Como se comentó durante la explicación del analizador léxico, el usuario siempre tiene la opción de generar un fichero LDV, usado por el programa de monitorización **SiMon**. Sin embargo, si el usuario en cualquier momento desea generar nuevamente este fichero a partir de las estructuras de datos en memoria, sin necesidad de recurrir al análisis léxico, no tiene más que pulsar el icono asociado a esta función, que tendrá como consecuencia la llamada al procedimiento ***genldv_proc()***.

Esta función acudirá a la librería desarrollada, y que recibe el nombre de ***genldv.c***, en la cual se mantienen todas las funciones necesarias para analizar las estructuras de datos del sistema,

y generar el fichero de salida. Mediante esta función, se separa la generación de este fichero de la necesidad de la ejecución del analizador léxico. Ello es bastante útil dado que, el usuario tiene la opción de generar las estructuras de memoria, no a partir del análisis léxico del lenguaje dado por el usuario, sino a partir de un fichero de disco.

5.5.3.8 Generación del fichero de información.

También se proporciona al usuario la capacidad para, a partir de las estructuras de datos que se mantienen en memoria, generar un fichero de información ASCII en el cual se muestre toda la descripción que el usuario ha realizado del sistema, dando información por tanto para cada elemento descrito, proporcionando información de la cantidad de memoria ocupada por ese elemento, en valores absolutos y en valores porcentuales, respecto de la memoria total requerida por el sistema.

La ejecución de esta función se realiza a través del procedimiento ***info_proc()***, el cual conducirá al usuario a la generación de este fichero de salida, que lo construirá a partir del nombre raíz, añadiéndole la extensión apropiada para este tipo de ficheros.

5.5.3.9 Compilación de fichero fuente.

A partir de la descripción dada por el usuario en el lenguaje LDAP interpretado, se ha comentado en repetidas ocasiones que se puede generar automáticamente un fichero fuente en lenguaje C, que recoja esta misma descripción, pero con el lenguaje LDAP compilado. Una vez que disponemos de este fichero fuente, se da al usuario la posibilidad de ordenar su compilación con las librerías adecuadas que han sido desarrolladas a tal efecto, generando un fichero ejecutable cuya ejecución tendrá por consecuencia la generación de estructuras de datos en memoria, así como el almacenamiento de las mismas en fichero binario (conteniendo toda la información contenida en ésta), así como demás ficheros de complemento.

La única previsión de error que hace la función ***comp_proc()***, encargada de ordenar la compilación del fichero C, es la de verificar que realmente existe este fichero. Para ello hace uso de la función ***stat()*** de la librería del C, la cual rellena una estructura de datos de tipo ***struct stat*** (definida en el compilador C) con información relativa al fichero indicado. Si este fichero no existe, devolverá un valor negativo que denunciará el error encontrado.

En cambio, si no se produce ningún contratiempo, se pasará a la ventana TTY de comandos la orden de compilación del fichero. Para ello se hará uso de la función ***ttysw_input()***, a la cual se le indicará una cadena del tipo:

bb nom_fich

donde: '*nom_fich*' es el nombre, sin extensión, del fichero fuente a compilar. Por otro lado, '*bb*' es el nombre de un fichero de procedimientos que incluye la llamada al compilador con las librerías y parámetros necesarios.

5.6 SiMon.

Tal como de explicó en el capítulo 3, SiMon es la herramienta encargada de la presentación gráfica de las arquitecturas objeto de estudio. En este apartado se realiza una descripción detallada de las distintas tareas encargadas a SiMon. En resumen podríamos decir que esta herramienta realiza cuatro actividades: lectura e interpretación de los comandos de los lenguajes LDV y LIV, gestión de todas las variables internas y realización de las transformaciones gráficas necesarias para la correcta visualización de las distintas arquitecturas y sus variables.

5.6.1 Intérprete del Lenguaje LDV.

Basándose en las instrucciones del lenguaje LDV, podrán ser descritos sistemas completos de una forma estructurada y respetando los niveles jerárquicos implícitos en ellos. A partir de la descripción se ha de generar una estructura de datos que sea reflejo del sistema. Esta estructura almacena toda la información referente al sistema descrito y las variables que contiene. Con ella se conocerán todos los elementos gráficos individuales, y como han de ser representados, el número de diseños (y ventanas) que contiene. Además, toda la información ha de ser accesible de forma simple.

Han de quedar reflejadas las relaciones entre elementos de nivel jerárquico inferior, las cuales constituyen elementos de niveles jerárquicos superiores. En definitiva, toda la información que nos puede ser útil para trabajar con la descripción deberá estar contenida en esa estructura de datos.

La ventaja aportada por la estructura de datos adecuada, es un acceso simple y rápido a los datos; la posibilidad de tener presente de forma inmediata toda la información que se requiera. Gracias a ella las funciones de dibujo de la representación del sistema han de trabajar de forma muy rápida, y con el mínimo esfuerzo de cálculo posible. Además, todas las variables han de tener su área reservada y ser localizables de forma rápida y simple.

La estructura de datos no puede limitar los niveles jerárquicos posibles en la descripción, y ha de adaptarse de forma óptima a cada descripción. De antemano no podrán ser limitados el número de elementos de cualquier tipo. Esto implica que la memoria será reservada en el momento que se precise y se ha de liberar cuando ya no sea necesaria. La potencia del lenguaje de descripción está en la facilidad para jerarquizar la descripción, y por ello no deberá ser limitada.

5.6.1.1 Estructuras de datos asociadas a las instrucciones del LDV.

Se pasa a describir a continuación, la estructura de datos que se ha utilizado para el almacenamiento de la descripción, partiendo de las declaraciones en LDV. Esta estructura tendrá forma de árbol, si bien, existirán enlaces entre nodos no relacionados directamente. El por qué se considera a la estructura un árbol, cuando existen enlaces entre los nodos no relacionados directamente, es porque estos enlaces no son parte de la estructura básica, sino información contenida dentro de los nodos, que indica donde están las descripciones usadas.

Dicho esto, para conocer el formato de la estructura, previamente dividiremos las instrucciones del lenguaje de descripción en tres tipos:

TIPO 1: ***BLOCK, SHEET, CELL*** y ***NETLIST***. *CELL*, es el elemento básico en el nivel jerárquico. *BLOCK*, es un nivel jerárquico superior y puede contener a otros *BLOCK*, en diferentes niveles jerárquicos. *SHEET* es idéntico a *BLOCK*, en cuanto a descripción, excepto que *SHEET* genera una representación gráfica y además los elementos puestos dentro de un *SHEET*, son elementos finales . Dicho de otra forma, un *BLOCK* sólo es una definición y un *SHEET* es una declaración para un elemento, por lo que requieren reserva de memoria para sus variables.

TIPO 2: ***PLACE*** y ***ARRAY*** son de una línea, pero hacen referencia a elementos ya definidos.

TIPO 3: Aquí hemos definido dos subtipos:

1) ***PIN*** e ***INT*** nos describen una variable elemental, y por ello han de tenerse en cuenta, puesto que el acceso a variable se hará según estén declaradas estas instrucciones.

2) **El resto de instrucciones** contienen información constante y simple.

Las instrucciones no sólo contienen información por su declaración, sino que también la contienen por la posición donde es declarada. Por ejemplo, una variable, que forme parte de una célula, elemental, puede ser replicada varias veces, según lo sea la célula, y es un elemento que pertenece a esa célula.

Los nodos de información de la estructura tienen formatos dispares según el tipo de dato que almacenen. La estructura refleja en su forma de árbol cómo es el sistema descrito. Los nodos que representan las instrucciones *TIPO 3*, tienen subestructuras hijas que contienen su declaración, y que, en principio pueden contener cualquier tipo de elementos. Así mismos, los nodos *TIPO 2* apuntan a descripciones de elementos.

Todos los nodos de información tienen una cabecera común (Fig. 5.5), que identifica el tipo de nodo al que pertenece. Reflejado por la declaración en lenguaje C:

```
typedef struct {
        int Clase;     /* tipo de NODO */
        void * info;   /* siguiente nodo del mismo tipo */
        void * link;   /* siguiente nodo de otro tipo  */
} Generic;
```

Una representación esquemática de esta cabecera se muestra en la Fig. 5.5 , la cual nos muestra de forma gráfica la declaración anterior. Todos los nodos contienen información heterogénea, y su estructura es diferente, según el tipo de dato que representen. De esta forma podemos forma una estructura con nodos heterogéneos. Cada nodo individual contendrá la información propia al tipo a que corresponda, más la cabecera que se ha mostrado. Esta cabecera ha de estar ubicada en algunas posiciones fijas, para poder reconocerla independientemente al tipo de nodo que sea el que se está tratando. Esto se muestra en la Fig. 5.6.

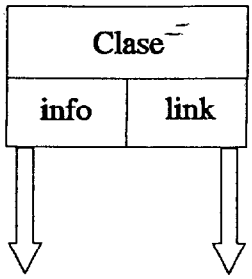

Fig. 5.5 Cabecera de un nodo LDV con sus clase y dos punteros.

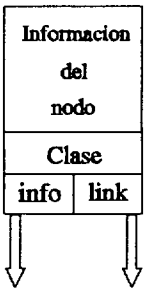

Fig. 5.6 Estructura general de un nodo LDV.

En este caso en particular, la estructura se ordena de tal forma que al puntero *info* se le enlaza el siguiente nodo de la misma clase, y al puntero *link* se le enlaza un nodo de la clase siguiente. Así, en la Fig. 5.7 tenemos una primera aproximación a la estructura que se genera. Sin embargo, esta estructura solo contiene nodos *TIPO 3*. Si hubiera nodos *TIPO 2* entonces habría que añadir punteros cruzados en medio de la estructura, partiendo cada puntero del área de información de cada nodo. Los nodos *TIPO 2*, además, generarán subestructuras, completas, que estarán ligadas al campo de datos, a través de punteros, tal como se muestra en la Fig. 5.8

El sistema, tan solo deberá conocer el primer nodo de la estructura, que en la figura se refleja como el puntero original *RAIZ*, y que en el código fuente es la variable global **Root**, la cual mantiene al primer nodo. Partiendo de este nodo, podemos recorrer toda la estructura, y de forma simple encontrar cualquier información que estemos buscando.

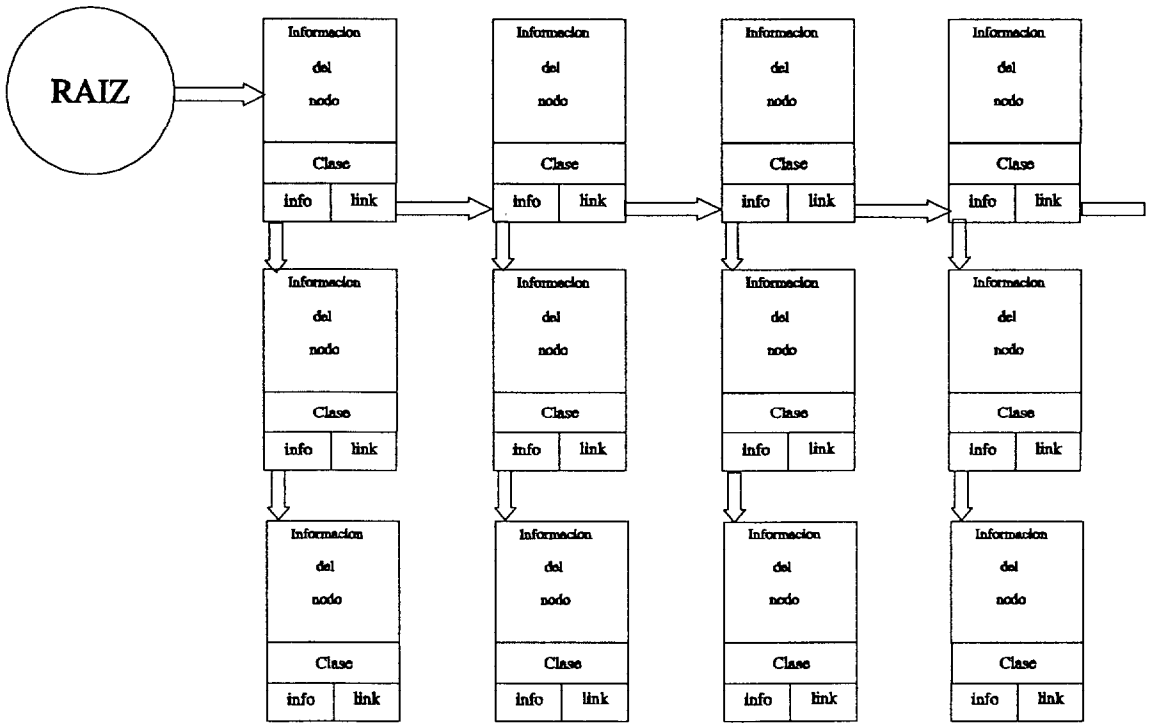

Fig. 5.7 Estructura elemental de nodos TIPO3 en la que cada columna está formada por nodos de la misma clase.

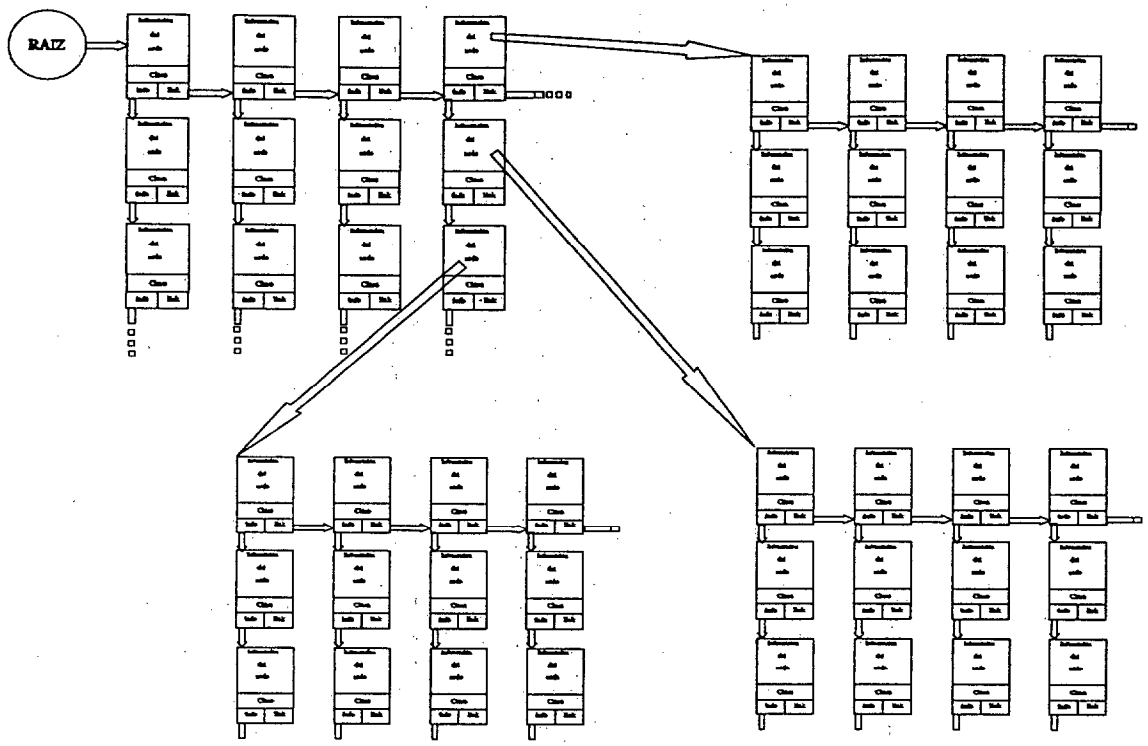

Fig. 5.8 Estructura completa de nodos TIPO 2 enlazados mediante punteros con estructuras elementales TIPO 3.

5.6.1.2 Descripción de los nodos de información.

Se han diseñado varios tipos de nodos para adaptarse a los diferentes tipos de instrucciones de que se dispone. Algunas instrucciones del LDV que contienen información similar han sido agrupadas en un mismo tipo de nodo.

String:

Que contiene como información un texto. Con este tipo de nodo se representan las instrucciones *DATE, AUTHOR, VERSION, LIBRARY.*

```
typedef struct {
        Generic *ID;            /* Cabecera */
        char *texto;            /*Información*/
}
```

Level:

Contiene el número de nivel, los niveles de rojo, azul y verde que, describe el color de este nivel, un bitmap que contiene el patrón del rellenado, una descripción textual y un byte de información reservado. Representa a una instrucción *LEVEL.*

```
typedef struct {
        Generic *ID;            /* Cabecera */
        byte nivel;             /* Nivel asociado */
        byte red;               /* Componente rojo */
        byte green;             /* Componente verde */
        byte blue;              /* Componente azul */
        byte status;            /* Reservado */
        Pixmap px;              /* Patrón de rellenado, si existiese */
        char *descripcion;      /* Nombre asociado */
} Level;
```

Model:

Contiene dos textos, un nombre completo de archivo y el tipo de modelo que contiene ese archivo. Representa a una instrucción *MODEL*.

```
typedef struct {
        Generic *ID;            /* Cabecera */
        char *Model;            /* Tipo de modelo */
        char *path;             /* archivo que lo describe */
        Frame fr;               /* ventana que los describe, si la tuviese asociada */
        int tipo;               /* Tipo de modelo reconocido */
} Model;
```

Cell:

Representa a una instrucción *CELL*. Contiene su nombre, su dimensión, el tamaño en bytes que ocupan sus variables y un puntero a una subestructura que contiene la descripción de la célula.

```
typedef struct {
        Generic *ID;            /* Cabecera */
        char *nombre;           /* Nombre de la célula */
        int Max_X;
        int Max_Y;              /* Ambito gráfico de la célula */
        void *Info;             /* puntero a la estructura de la descripción de la célula */
        int Size;               /* Tamaño total de sus variables */
} Cell;
```

BloDes:

Representa a las instrucciones *BLOCK* y *SHEET*. Contiene el nombre, un puntero a la descripción, la cantidad de memoria requerida por las variables del *BLOCK/SHEET*, y el resto de los parámetros son usados específicamente por la instrucción *SHEET*. Estos son: un puntero a un área de memoria reservada para almacenar las variables del diseño, los punteros a la ventana principal y ventana gráfica (*frame* y *canvas*, en términos de *OpenWindows*), un puntero que apunta una subestructura, que almacena los parámetros gráficos del diseño, una lista de transformaciones gráficas, para mantener los escalados y desplazamientos del gráfico, y una lista que nos indica qué niveles son visibles o no. Esta es la estructura de datos más compleja, con diferencia.

```
typedef struct {
        Generic *ID;            /* Cabecera */
        char *nombre;           /* Nombre del bloque */
        int Size;               /* Espacio que ocupan sus variables */
        int Max_X;
        int Max_Y;              /* Dimensiones gráficas */
        void *Mem;              /* Espacio de memoria reservado para el almacenamiento de sus variables */
        Frame fr;               /* Ventana asociada al SHEET */
        Canvas cv;              /* Ventana gráfica asociada al SHEET */
        GrParam *gp;            /* Puntero a una estructura específica, utilizada para almacenamiento de información gráfica */
        CmList *List;           /* Lista de transformaciones gráficas que lleva asociado el SHEET */
        StList *LevCtl;         /* Lista que contiene la información de visibilidad para cada nivel */
} BloDes;
```

Pin:

Representa a una instrucción *PIN*. Contiene el nombre, el tipo de variable, el sentido, el nivel y su posición dentro de la célula, o el lugar donde se encuentre. También contiene un offset, que nos indica donde se almacena su variable relativo al inicio de la memoria asociada al elemento al cual pertenece.

```
typedef struct {
        Generic *ID;            /* Cabecera */
        char *nombre;           /* Nombre */
        byte tipo;              /* Identificador de tipo de variable */
        byte sentido;           /* Identificador del sentido del PIN */
        byte nivel;             /* Identificador del nivel al que está asociado */
```

```
        int x;
        int y;          /* Coordenadas gráficas en las que se encuentra */
        int offset;     /* Dirección relativa de almacenamiento */
} Pin;
```

Int:

Representa a una instrucción *INT*. Contiene el nombre, el tipo de variable, el nivel y el offset de su variable relativo al inicio de la célula.

```
typedef struct {
        Generic *ID;        /* Cabecera */
        char *nombre;       /* Nombre */
        byte tipo;          /* Identificador de tipo de variable */
        byte nivel;         /* Nivel asociado a la variable */
        int offset;         /* Dirección relativa de almacenamiento */
} Int;
```

LinRec:

Representa a las instrucciones *LIN* y *REC*. Contiene el nivel y las coordenadas de la línea o rectángulo que representa.

```
typedef struct {
        Generic *ID;        /* Cabecera */
        byte nivel;         /* Nivel asociado */
        int x0,y0;
        int x1,y1;          /* Coordenadas de dibujo */
} LinRec;
```

Box:

Representa a una instrucción *BOX*. Contiene el nivel del borde y el nivel del rellenado y la información de la caja que representa.

```
typedef struct {
        Generic *ID;        /* Cabecera */
        byte borde;         /* Nivel asociado al borde */
        byte fondo;         /* Nivel asociado al fondo*/
        int x0,y0;
        int x1,y1;          /* Coordenadas de dibujo */
} Box;
```

Pol:

Representa a una instrucción *POL*. Contiene el nivel del borde y del rellenado, el número de puntos que tiene el polígono y dos punteros a las coordenadas x e y del polígono.

```
typedef struct {
        Generic *ID;        /* Cabecera */
        byte borde;         /* Nivel asociado al borde */
        byte fondo;         /* Nivel asociado al fondo*/
        int npuntos;        /* Numero de vértices */
        int *x;
        int *y;             /* Punteros a vectores de coordenadas */
} Pol;
```

LabelTxt:

Representa a las instrucciones *LABEL* y *TXT*. Contiene el texto, nivel, escala del texto y su posición dentro de la célula. Además contiene un offset, para indicar en que posición está la variable que pueda estar asociada a *LABEL*.

```
typedef struct {
        Generic *ID; /* Cabecera */
        char *texto;        /* Texto */
        byte nivel;         /* Nivel asociado */
        int escala;         /* Factor de escala del texto */
        int offset;
        int tipo;           /* Elementos para la identificación de la variable que pudiera estar asociada a LABEL */
        int x,y;            /* Coordenadas de dibujo */
} LabelTxt;
```

Array:

Representa a la instrucción *ARRAY*. Contiene el nombre del array, un offset de dibujo inicial, el número de filas y columnas, el incremento entre filas y columnas para dibujo, el tamaño que ocupan las variables del array y un puntero que apunta a la descripción del elemento que se replicará.

```
typedef struct {
        Generic *ID;            /* Cabecera */
        char *nombre;           /* Nombre asociado */
        int Max_X;
        int Max_Y;              /* Dimensiones gráficas */
        int x;
        int y;                  /* Offset inicial de dibujo */
        int n_filas;
        int n_columnas;         /* Número de filas y columnas */
        int inc_filas;
        int inc_columnas;       /* Incremento gráfico para el dibujo del ARRAY */
        int Size;               /* Espacio de almacenamiento de variables */
        void *celula;           /* Puntero al elemento que replica */
} Array;
```

Place:

Representa a la instrucción *PLACE*. Contiene el nombre, la posición, la rotación, un puntero que apunta a la descripción del elemento que se va a ubicar, y un offset que nos indica donde se ubicarán las variables del elemento que se ubica.

```
typedef struct {
        Generic *ID;            /* Cabecera */
        char *nombre;           /* Nombre asociado */
        int Max_X;
        int Max_Y;              /* Dimensiones gráficas */
        int x;
        int y;                  /* Offset de dibujo */
        int grados;             /* Angulo de rotación */
        int offset;             /* offset de las variables que lleva asociada */
        void *elemento;         /* Puntero a la declaración del elemento que posiciona */
} Place;
```

Nodo:

Representa a la instrucción *NODE*. Contiene el nombre asociado al nodo al que apunta, y un vector de punteros a las direcciones de memoria de las variables a las que está asociado. El último elemento de este vector es un puntero nulo.

```
typedef struct {
        Generic ID;             /* Cabecera */
        char *nombre;           /* Nombre del nodo */
        char **varlist;         /* Lista de variables */
} Nodo;
```

Netlist:

Representa a la instrucción *NETLIST*. Contiene tan solo un puntero a una descripción, en la cual habrá una lista de nodos asociados.

```
typedef struct {
        Generic ID;             /* Cabecera */
        void *descripcion;      /* Descripcion */
} Netlist;
```

Todas las instrucciones (menos *END*), generan un nodo de información, que se inserta en la estructura en un sitio determinado. Ya se comentó el hecho de que existían tres tipos de instrucciones. Ahora cobrará significado completo esta descripción.

TIPO 1: *SHEET, BLOCK, CELL.* Contienen como descripción una estructura completa que contiene todos los elementos de su definición (hasta la instrucción *END*). Así el siguiente ejemplo:

```
BLOCK BL
        LIN...
        REC...
        LIN...
        PIN...
        PIN...
        DATE...
        REC...
END
```

Esta declaración generará una estructura tal como la que se puede observar en la Fig. 5.9, en ella se observa como la descripción asociada al *BLOCK BL* es una subestructura completa.

TIPO 2: *ARRAY y PLACE.* Toman como base un elemento ya descrito, por lo que contienen un puntero que referencia al elemento (sea célula o bloque). Así:

PLACE P1 ... ELEMENTO

Generará una estructura tal como la que se ve en la Fig. 5.10. En general,un elemento puede ser referenciado cuantas veces sea necesario.

TIPO 3: Son el resto de elementos. Tan sólo van insertados en la estructura.

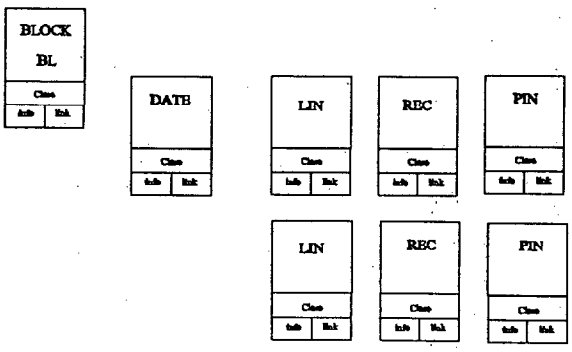

Fig. 5.9 Ejemplo de estructura generada por una descripción TIPO 1.

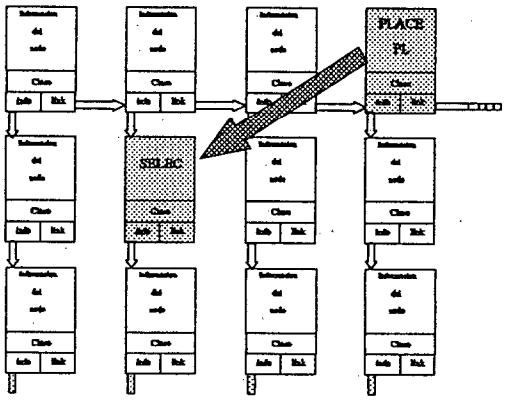

Fig. 5.10 Ejemplo de referencia desde elemento TIPO 2.

5.6.1.3 Descripción del interprete del lenguaje LDV.

Antes de entrar en detalle acerca de como se interpreta las instrucciones o como se genera los nodos de información o las áreas de memoria, hemos de tocar el tema del formato del archivo. El formato primario es:

<PALABRA CLAVE> <LISTA DE PARAMETROS OBLIGATORIOS> [<LISTA DE PARAMETROS OPCIONALES] [<COMENTARIOS>]

En principio, una línea debe comenzar con una palabra clave seguida de una lista de parámetros obligatorios y terminada con una lista de parámetros opcionalmente o con comentarios o nada. En el traductor que se ha implementado se permite la utilización de espacios en blanco o tabuladores en número ilimitado (Estos, tan solo no pueden encontrarse en medio de las palabras claves). Sólo se pueden colocar un comando por línea y el carácter de nueva línea será interpretado como finalizador de comando. Se admiten las líneas en blanco. Los separadores son: el espacio en blanco, el tabulador, los dos puntos y el punto y coma. No se ha incluido la coma como separador por el hecho de que algunos nombres pueden llevar comas insertados. Esto es debido, a que podríamos definir un *ARRAY* manualmente dando su nombre con los índices separados por comas pero definiendo individualmente cada elemento. Los nombres pueden contener cualquier ristra de caracteres que no contenga separadores.

Una vez especificado estrictamente el formato de entrada al traductor, nos podemos plantear el método utilizado para atacar el problema. De entrada vemos que la estructura del lenguaje está orientada a líneas. Por ello, nos debemos plantear inicialmente una función para leer líneas. Tras ésto ejecutamos una función que coja esta línea y la divida en primitivas. Ella identificaría cada uno de los elementos individuales identificados entre separadores respetando las comillas. Con este trabajo hecho ya se puede pasar a interpretar la línea.

Al interprete en sí, se le pasaría la lista de todos los identificadores reconocidos. En principio, para un sistema orientado a archivos al lector de líneas se le debería pasar un descriptor de archivos el cual contiene al archivo en proceso. En la práctica, al lector de líneas se le pasa una lista de descriptores de archivos con todos los archivos abiertos y en proceso. Esto es debido, al hecho de que se permiten archivos incluidos. De esta forma, el lector de líneas estaría procesando el último archivo de la lista. Si encontrara el fin del archivo, cerraría ese elemento y pasaría a procesar el siguiente. Cuando se encontrara una instrucción de archivos incluidos, tan sólo habría que abrir el archivo indicado en el comando. El nuevo descriptor de archivos generado sería añadido a la lista. Esta estructura se puede observar en la Fig. 5.11. El nodo elemental de esta lista se muestra en la Fig. 5.12. De esta forma, la próxima vez que llamemos al lector de líneas éste a su vez leerá líneas del nuevo archivo. Con este sistema, podemos insertar todas las declaraciones contenidas en un archivo en un punto. El nodo elemental descriptor de archivo contendría, el descriptor del archivo que tiene asociado, su nombre y el número de líneas que contiene.

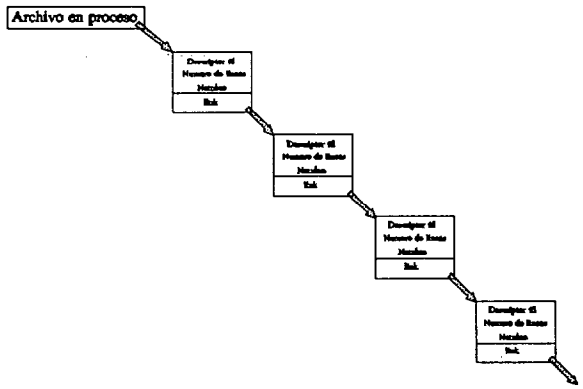

Fig. 5.11 Lista de descriptores de archivo.

Al lector de líneas se le ha asignado además, otra tarea importante. Esta tarea es la de reconocer las líneas de comentarios.

Tras la lectura elemental de la línea deberíamos tratar de reconocer los elementos que contiene esa línea. Para ello, partimos de la base de que el lector de línea identifica la líneas de comentario y no las devuelve. Tras el lector tenemos una lista de líneas con significado real (sin líneas de comentario). Tras ésto, se llamará a la función divisora de líneas que genera un vector de ristras de caracteres conteniendo todos los elementos identificadores

encontrados. Como parece lógico, en esta función se identifica los entrecomillados y no se divide. Al final del proceso esta función devuelve un vector (ristra de caracteres) conteniendo cada una de las partes, indicando el final de dicho vector mediante el elemento nulo. De esta forma, se puede ir leyendo subelemento a subelemento hasta encontrar el final de la línea (que sería el elemento nulo). Como apoyo devuelve el número de parámetros encontrados.

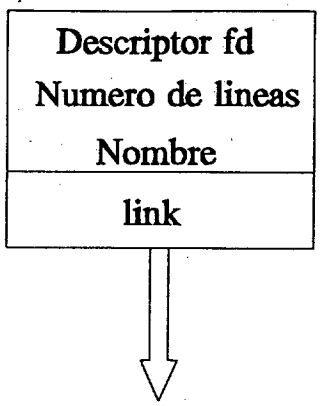

Fig. 5.12 Nodo elemental descriptor de archivo.

Como última función de apoyo se describe otra función que se llama intérprete de línea. En realidad esta función lo que hace, es identificar el primer elemento de la línea ya partida con un patrón de palabras claves. Si encuentra que el primer parámetro coincide con alguna palabra clave, devuelve un identificador indicándolo. Si, por otro lado la línea que se le pasa está vacía, es decir, no existe ningún elemento en el vector de parámetros, devuelve el carácter nulo. Si, el primer parámetro contiene texto, pero no se identifica como una palabra clave, devuelve un identificador indefinido. Cuando se dice que se devuelven identificadores en realidad lo que se devuelve son números enteros. Estos números enteros son constantes en lenguaje C, definidas de la forma:

```
#define UNDEF   -1
#define EMPTY    0
#define DATE     1
#define AUTHOR   2
```

Con estas definiciones podemos programar de forma simbólica. En realidad, puede parecer poco útil utilizar etiquetas para codificar números simples, pero, lo que se está haciendo es codificar identificadores con números enteros. Este tipo de definiciones hace el programa C generado más legible, más flexible y facilita las modificaciones que se deseen hacer.

Con las tres primitivas que se han descrito, es decir, lector de líneas, divisor de líneas e intérprete de líneas, podemos construir un analizador sintáctico. Este analizador procesa el archivo de entrada y genera lista de parámetros diciendo el tipo de comando que contiene. Además, en la implementación práctica, al analizador sintáctico se le ha asociado la tarea de identificar líneas correctas o no. En la práctica, lo que hace es ver si el número de parámetros concuerda con el tipo de declaración que corresponde. Los parámetros sólo se pueden limitar en cuanto a su número mínimo, es decir, una declaración debe contener un número de parámetros obligatorios, y el resto de parámetros son opcionales. Para ello, lo que se hace es ver si el número de parámetros que hay es menor que el mínimo especificado como obligatorio. Si ésto es así, se devuelve un error. Así el analizador sintáctico que se ha implementado tiene la declaración C que sigue:

```
int Sintáctico(tok)
char *tok[]; /* tok: Lista de argumentos de salida */
{
        char s[MAX_STRING];
        int p,t,i;

        /* Inicializa la lista de argumentos de salida */
        tok[0]=NIL;
        /* Lee una línea y devuelve error en caso de fallo */
        if ((p=LeeLinea(s))!=0) return p-2;
        /* Si hay línea la parte */
        p=strsplit(s,tok);
```

```
    /* Si la línea está vacía devuelve vacío */
    if (p==0) return EMPTY;
    /* Trata de reconocer alguna palabra clave */
    t=token(TokenList,tok);
    /* Si no identifica ninguna palabra clave devuelve el error correspondiente */
    if (t==UNDEF) return -4;
    /* Si no hay suficientes argumentos devuelve el error correspondiente */
    if (p<NArg[t-1]) return -5;
    /* Devuelve el identificador encontrado */
    return t;
}
```

Este listado nos muestra el analizador sintáctico utilizado. En caso de error se devuelve un código que identifica dicho error. Con estas funciones podemos abstraernos del problema de identificar las líneas y de leer archivos. Basándonos en ellas, tan sólo hay que hacer llamadas sucesivas al analizador sintáctico y éste nos irá devolviendo los identificadores correspondientes y las listas de parámetros asociados a ellos. En caso de encontrar un error en la interpretación, como ya se ha dicho, también nos indicara el tipo. Así, podríamos escribir el traductor de forma mucho más simple.

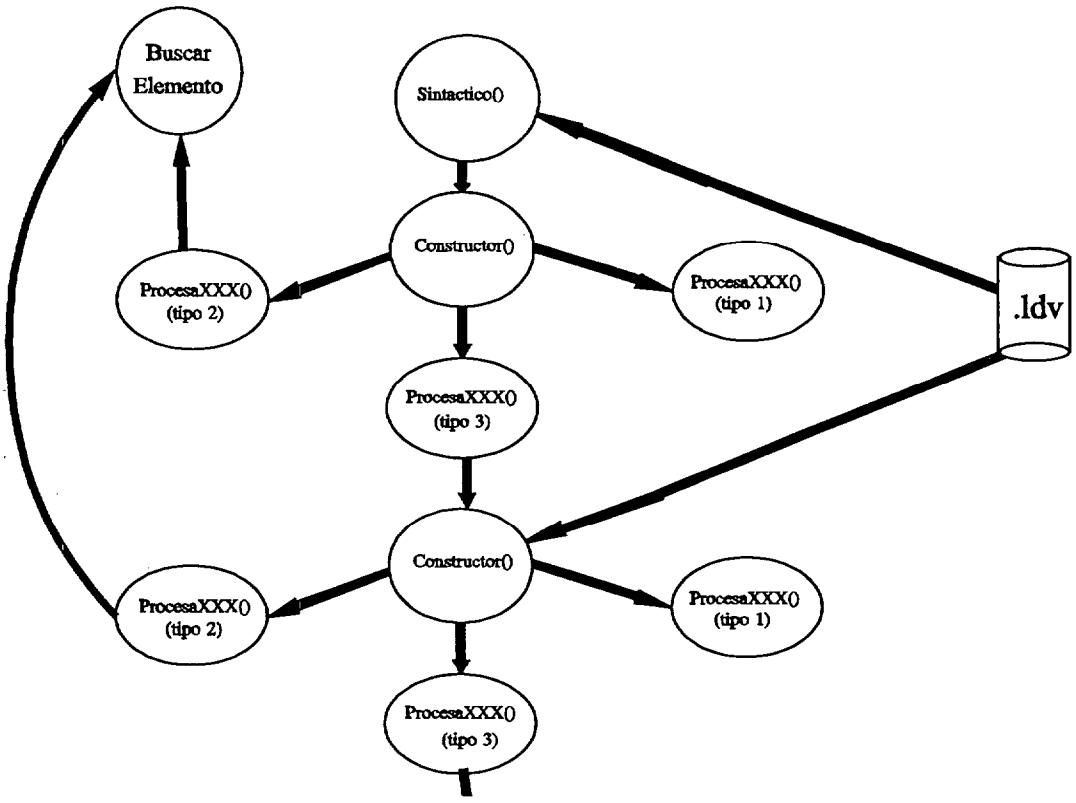

Fig. 5.13 Estados del interprete del lenguaje LDV.

En vista de que el lenguaje de descripción visual puede tener ilimitados números de niveles jerárquicos, hemos de implementar una función para la traducción que sea recursiva. En general, se puede considerar la traducción de una descripción como un problema global. De esta forma, la traducción de la descripción de un *BLOCK* a un *SHEET* o una *CELL*, es un problema de la misma especie que traducir una descripción general (he aquí el principio que nos introduce en la recursividad). Por ello, se plantea una función que va construyendo la estructura a medida que va leyendo el archivo y que, cuando encuentre el identificador de final de descripción (tipo *END*) finalice devolviendo un puntero a la raíz estructura de datos que ha generado. Así, podríamos generar descripciones de elementos como los mencionados, simplemente llamando recursivamente a la función que genera la descripción. El valor que devolvería esta función (puntero a la raíz de la descripción generada) se asociaría al nodo de información del elemento que sea el padre de esa descripción.

Un esquema de bloques de la forma de funcionamiento de la traducción sería el que se muestra en la Fig. 5.13. Este esquema nos muestra como funciona la traducción. En ella se aprecia una función central denominada Constructor(). Se puede ve al analizador sintáctico como un bloque que va leyendo elementos del archivo de entrada y va generando lista de tokens. En general, se puede considerar a la función Sintáctico() como la suministradora de instrucciones para ser traducidas. Se puede observar, que el punto central es la función Constructor() que genera llamadas a la función ProcesaXXX(). En sí, Constructor() identifica el tipo de elemento o palabra clave que genera Sintáctico() y reconoce que función de procesado hay que llamar. Así mismo, existe una función de procesado para cada tipo de datos, habiendo alguna que procesa varios tipos de datos. En general, se ve que Constructor() llama a tres tipos de elementos que serían los procesos correspondientes a cada tipo de nodo que se ha comentado. Los nodos de TIPO 1 no generan ninguna llamada, tan sólo se procesa la información que le pasa Constructor() y se devuelve el nodo procesado. Los nodos de TIPO 2 hacen una llamada a la función de búsqueda del elemento que puedan tener asociado y devuelven el nodo que han generado. Los nodos de TIPO 3 llaman recursivamente a la función Constructor() indicando el tipo de padre que es, es decir, el contexto en el que están siendo llamadas. Al ejecutarse Constructor(), éste procesa la información que viene de Sintáctico() generando una estructura de llamadas que es idéntica a la del constructor raíz. Tras acabar esta función, devuelve un puntero a la raíz generada que es utilizada por el ProcesaXXX() de TIPO 3 para conocer la descripción que lleva asociada ese elemento. Con esta estructura no existe limitación del número de niveles jerárquicos que se generan, como ya se ha dicho.

En la Fig. 5.13 tan solo se muestran dos niveles (dos Constructor()), pero las llamadas recursivas a constructor no tienen límite.

Bajo el nombre ProcesaXXX() existen una serie de funciones especializadas en el proceso de cada instrucción. Así existen ProcesaDate(), ProcesaLevel() ... Constructor(), al identificar el tipo de instrucción que se está procesando, genera la llamada a cada tipo de ProcesaXXX(). Cada tipo de ProcesaXXX(), generará un nodo de información, con todos sus campos actualizados, según los argumentos de la instrucción en proceso, y la insertará en el punto correspondiente de la estructura. Así se va generando la estructura de datos.

Para insertar nodos, se ha de partir del primer nodo de la estructura. Mientras la clase del nodo que se va a insertar sea menor que el nodo que se va encontrando, se va caminando por los punteros *link*. Cuando se encuentre un nodo de la misma clase, se empieza a caminar por los punteros *info*. El nodo que se va insertar, se enlaza del puntero *info* del último nodo que se encuentre.

Si se encuentra un nodo de clase superior al que se pretende insertar, o se llega al final recorriendo la vía *link*, sin encontrar ningún nodo de clase igual o mayor al nuestro, quiere decir que nuestro nodo es el primero de esa clase que se trata de insertar. Esto implica, que hemos de asociar el puntero *link* del último nodo de clase inferior al nuestro a nuestro nodo, y a ese último nodo de clase inferior al nuestro, le asociamos en su puntero *link* la dirección de nuestro nodo. Si no existen nodos, nuestro nodo pasa a ser la estructura completa.

Las descripción de elementos que lleven descripción (*BLOCK*, *SHEET*, y *CELL*), se construyen, (como ya indicó) a partir de descripciones vacías. Cuando finaliza esta descripción, el puntero al primer nodo se asocia al puntero que nos diga donde va la descripción del elemento que estamos procesando.

5.6.2 Interprete del lenguaje LIV.

5.6.2.1 Proceso de comunicación mediante memoria compartida.

Las funciones de monitorización y control asignadas a *SiMon*, llevan implícitas unas tareas de comunicación con el mundo externo a ellas. Este mundo externo puede ser desde la propia estación de trabajo, donde se puede estar realizando una tarea en paralelo generadora de datos, hasta elementos externos a la estación y que se comuniquen con ella a través de sus puertos. Por otro lado, la tarea que genera los datos en la estación ha de ser independiente de la tarea central. Esto debe ser así porque si no, para cada aplicación particular se debería recompilar a *SiMon* con las funciones de obtención de datos apropiadas. Al ser *SiMon* independiente de la forma en que se obtienen los datos, para cada aplicación particular tan sólo se debe generar una pequeña tarea de intercambio de variables.

Para la implementación de estas primitivas de comunicación se ha utilizado el mecanismo de comunicación entre procesos utilizando *memoria compartida* de *UNIX*. Este elemento nos permite generar un área de memoria (similar a la que generaría la llamada malloc() a la que pueden acceder otros procesos de los que se estén ejecutando en la máquina. En la Fig. 5.14 se puede ver un esquema de como funciona esta *memoria compartida*. El sistema utilizado en *UNIX* para identificar un área de memoria compartida como única, y permitir el acceso desde el exterior a otro proceso identificando a que área se desea acceder es darle un número de identificación, número que es generado arbitrariamente por el proceso. Cuando un proceso abre un área de memoria compartida con un identificador que no se haya utilizado, el sistema crea ese área y devuelve un puntero a ella. Si el identificador que se utiliza para abrir la memoria compartida ya fue utilizado, el sistema conecta la nueva área con la ya creada, devolviendo un puntero a esa área.

Fig. 5.14 Esquema de funcionamiento elemental de la memoria compartida.

5.6.2.2 Funciones de comunicación.

Para realizar la comunicación con *SiMon* se han implementado una serie de funciones que nos permiten acceder de forma simple. Estas funciones están implementadas en un archivo de cabecera en lenguaje C denominado 'shmail.h '. Este archivo debe estar incluido en cualquier programa C que se desea que trasmita datos a *SiMon*. Contiene una lista de declaraciones y la implementación de ciertas funciones que hace elemental cualquier tipo de comunicación.

El área de memoria compartida utilizada para la comunicación tendrá una estructura predeterminada. La declaración de esta estructura puede verse al comienzo del archivo 'shmail.h'. Esta declaración es de la forma:

```
#define COM_SPACE      512

typedef struct {
        char status;
```

```
        int rdversion;
        int wrversion;
        int size;
        char mail[COM_SPACE];
} ShMail;
```

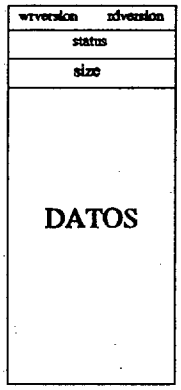

Fig. 5.15 Area de comunicación de la memoria compartida.

La estructura se puede observar que tiene cinco campos (Fig. 5.15). El primer campo denominado **status** se utiliza para almacenar información acerca del modo de funcionamiento de la comunicación. Se pueden utilizar ciertos bits para bloquear la comunicación en un sentido u otro. Los dos siguientes campos denominados **rdversion** y **wrversion**, son utilizados para sincronizar la comunicación. El campo **size** nos indica el tamaño del mensaje que está en el área de datos y el campo **mail[COM_SPACE]** es el área de datos. El protocolo de funcionamiento de la comunicación con esta estructura es muy simple. El campo *rdversion* almacena el número de identificación del último mensaje leído por el proceso lector, así mismo, el campo *wrversion* almacena el número del último mensaje escrito por el proceso escritor. Si *rdversion* es distinto a *wrversion*, significa que existe un mensaje en el área de datos (el tamaño viene indicado por el campo size). Si un proceso escritor deseara escribir podría hacerlo de dos formas. La primera forma, sin respetar el protocolo, sería con la siguiente secuencia: se escribe los datos en el campo *mail[COM_SPACE]*, luego se actualiza el tamaño del campo de datos en *size* y finalmente se incrementa el campo *wrversion* que nos indica la versión de datos que está disponible en el área de comunicación. Con este método no respetamos el protocolo en tanto en cuanto, si hubiera un mensaje que no haya sido procesado, sería anulado ya que se escribiría sobre él. La siguiente forma sí respetaría el protocolo. Esto quiere decir que el proceso escritor antes de escribir chequea si el último mensaje escrito fue leído. Esto se indica mediante el campo rdversion de tal forma que, el valor que almacena debe coincidir con el valor de wrversion. Si el campo de datos aún no ha sido leído, es decir, *rdversion* distinto a *wrversion*, no se puede escribir. Ante ésto, existen dos posibilidades: devolver un valor de fallo al usuario que ha llamado a la función o, esperar a que cambie el valor del campo.

El proceso lector tan sólo deberá leer el campo *rdversion* y compararlo con *wrversion*. Si ambos son distintos (existe un mensaje en el área de datos) se leería el número de elementos del campo *mail[COM_SPACE]* que nos indique el campo *size* y a continuación le daríamos a *rdversion* el valor que almacene *wrversion*, indicando así que el mensaje ha sido leído.

En la implementación de *SiMon* se han utilizado dos áreas de datos para establecer la comunicación con el mundo externo. Con estas dos áreas podemos hacer una transmisión full-duplex, utilizando una para transmitir y otra para recibir (Fig. 5.16). Las áreas de comunicación definidas se llaman **Send** y **Receive**. A través de *Send* podemos enviar mensajes a *SiMon* y éste nos lo enviará a través de *Receive*. Los identificadores de memoria compartida que se han utilizado se pueden ver antes de la declaración de estas dos áreas de memoria compartida, denominándose **SEND_KEY** y **RECEIVE_KEY**. A continuación de estas declaraciones iniciales se encuentra la declaración de las funciones de comunicación que se han implementado. Con estas funciones el proceso de comunicación se facilita bastante y es conveniente utilizarlas en todos los procesos que van a estar en comunicación con *SiMon* para respetar la uniformidad. Es importante establecer un estándar en las comunicaciones para

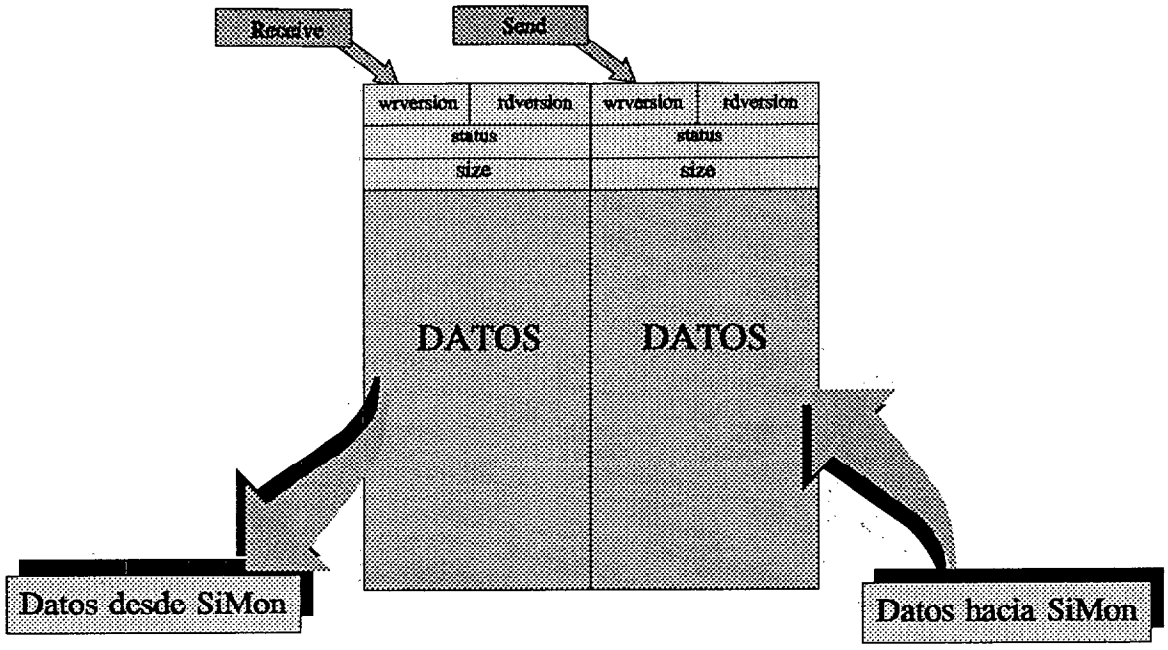

Fig. 5.16 Representación del mecanismo de comunicación entre procesos a través de la memoria compartida.

que el sistema funcione correctamente.

Las dos primeras funciones que se declaran son las que deben ser utilizadas para el establecimiento de la comunicación. Estas funciones son:

int ShMInit();

Esta función se utiliza para crear el área de comunicaciones. En realidad, esta función está aquí por el hecho de que en este archivo ('shmail.h'), se incluyen todas las funciones que se utilizan para la comunicación, aunque ésta no debe ser llamada por el usuario, y sólo *SiMon* es el que puede hacerlo para crear el área de datos. Además de crear el área de datos, esta función inicializa los valores que contiene.

int ShMConect();

Esta función se utiliza para conectarse al área de datos de *SiMon*. Esta es la función que se debe utilizar antes de empezar un proceso de comunicación con SiMon, encargándose de inicializar los punteros al área de datos (Fig. 5.17).

A continuación vienen tres funciones para la comunicación elemental con el área de datos:

*int ShMWrite(ShMail *p, char *buf, int size);*

Esta función se utiliza para transmitir datos sin respetar el protocolo de transmisión, es decir, si hubieran datos en el campo de datos que no hubieran sido leídos éstos serán anulados al hacer la llamada a esta función. En el

parámetro *p*, se le debe pasar un puntero al área de memoria compartida que se va a utilizar para transmitir datos (en ella se almacenan dichos datos hasta que son leídos). En el parámetro *buf*, se le debe pasar un puntero al área de datos que se quiere transmitir y por último en el parámetro *size* se debe pasar el número de elementos que se desea transmitir. Esta función devuelve el número de elementos transmitidos. En un caso estándar de comunicación con *SiMon*, a través de p se le debería pasar la variable global *Send* (Fig. 5.18).

int ShMWriteFr(ShMail *p, char *buf, int size);

Esta función es similar a ShMWrite(). La diferencia con ella es que respeta el protocolo, es decir, que en caso de que el área de comunicación no esté vacío no escribe nada, devolviendo cero. El valor cero devuelto puede ser utilizado por la función que la llamo para detectar que no se pudo escribir. Otra forma de detectar fallos en las dos funciones de escritura es ver si el número de elementos que han sido escritos coincide con el número de elementos que se enviaron para escribir (Fig. 5.19).

int ShMRead(ShMail *p, char *buf);

Esta función se utiliza para leer datos de un área de comunicación. El puntero al área de comunicación utilizada se le pasa a través del parámetro p. Los datos leídos son depositados en el área apuntada por el parámetro buf. Esta función devuelve el número de elementos leídos. En un caso estándar de comunicación con SiMon, a través del parámetro p se le debería pasar la variable global Receive.

Las tres funciones de comunicación escritas anteriormente, pueden ser utilizadas cuando sólo existe un proceso que se comunica con *SiMon*. En caso de existir varias fuentes de datos hacia *SiMon*, se plantea un problema grave y es que pueden estar escribiendo datos simultáneamente destruyéndose la información unos a otros. Para arbitrar el proceso de comunicación evitando colisiones, son implementadas además, otras funciones que manejan **semáforos**. En la práctica, los *semáforos* se han implementado utilizando los bits del campo *status* del área de comunicación. Se han definido dos bits, uno como semáforo de escritura y otro como de lectura. Para la gestión de estos bits se han escrito cuatro funciones que se detallan a continuación:

void LockWr(ShMail *p);

La misión de esta función es bloquear el semáforo de escritura.

void LockRr(ShMail *p);

Esta función se encarga de bloquear el semáforo de lectura.

void UnLockWr(ShMail *p);

Esta función desbloquea el semáforo de escritura.

*void UnLockRr(ShMail *p);*

Esta función desbloquea el semáforo de lectura.

Cuando un proceso desea entrar en su **zona crítica**, es decir, trata de procesar datos que pueden estar compartidos con otro proceso, debe respetar el siguiente protocolo de comunicaciones (Fig. 5.20):

1) Analizar si los bits del *semáforo* de la operación que desea hacer están ocupados. Si esto fuese así, debería esperar a que se desocuparan.

2) Cuando los bits del *semáforo* buscado estén libres, el proceso los ocupa, entrando en su **zona crítica**. A partir de ahora puede considerar el área compartida como suya y realizar el procesado que se necesario.

3) Cuando acaba de procesar el área compartida, se liberan los bits del *semáforo*, saliendo el proceso de su **zona crítica**.

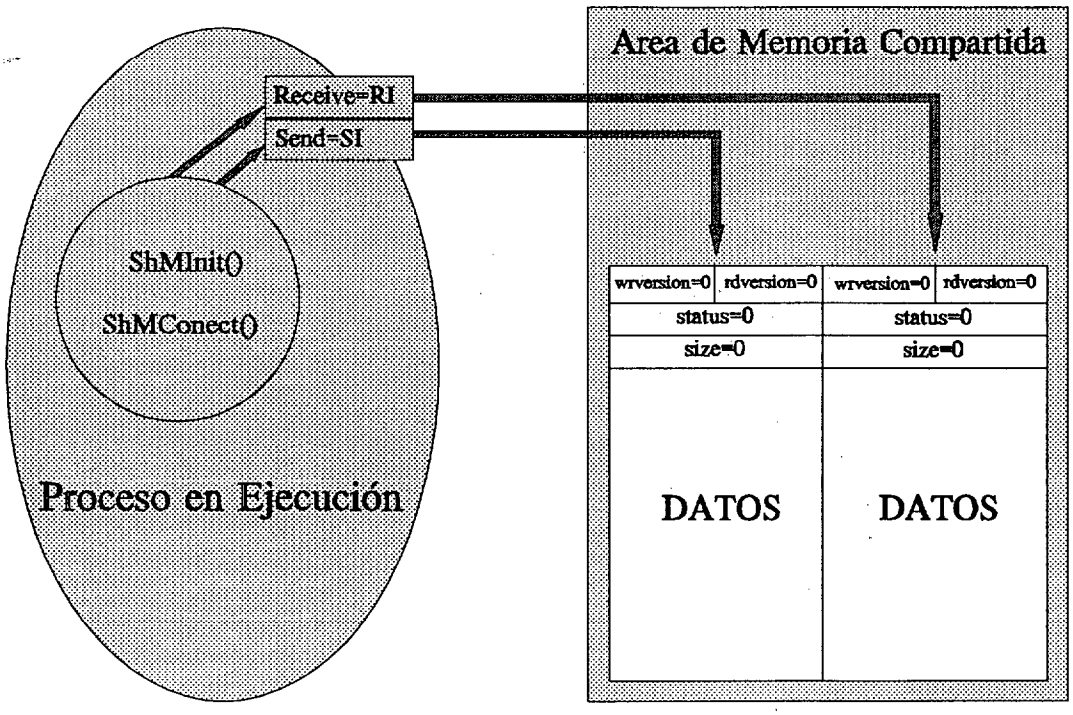

Fig. 5.17 Esquema de la conexión con el área de memoria compartida.

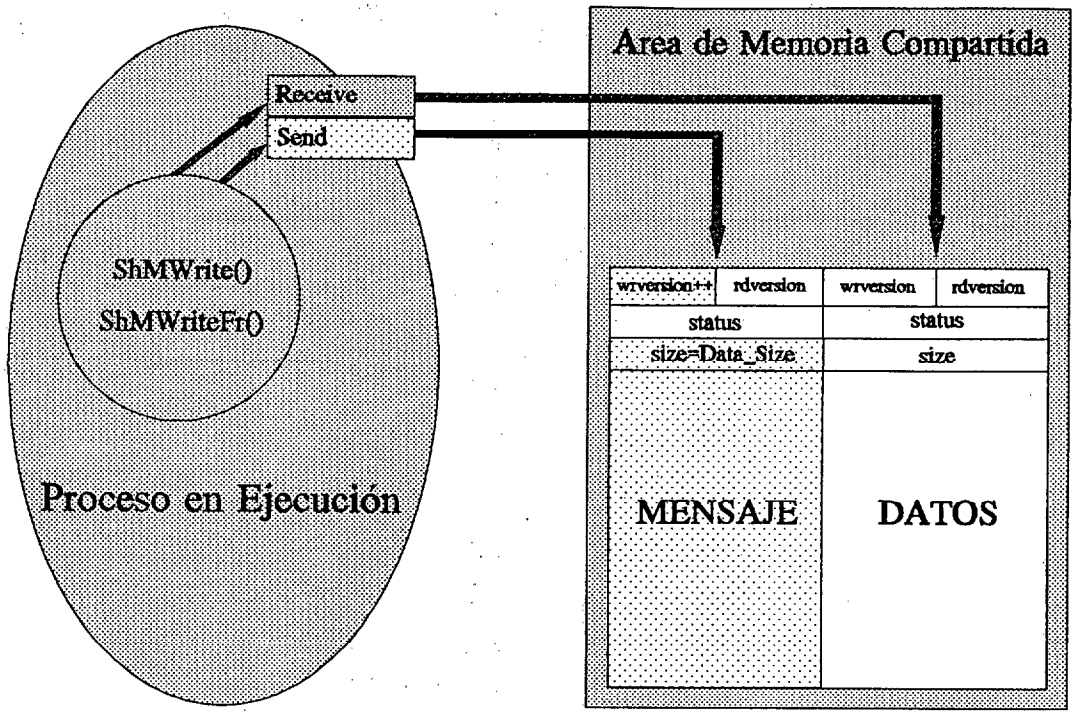

Fig. 5.18 Esquema de la transmisión de datos a través de memoria compartida.

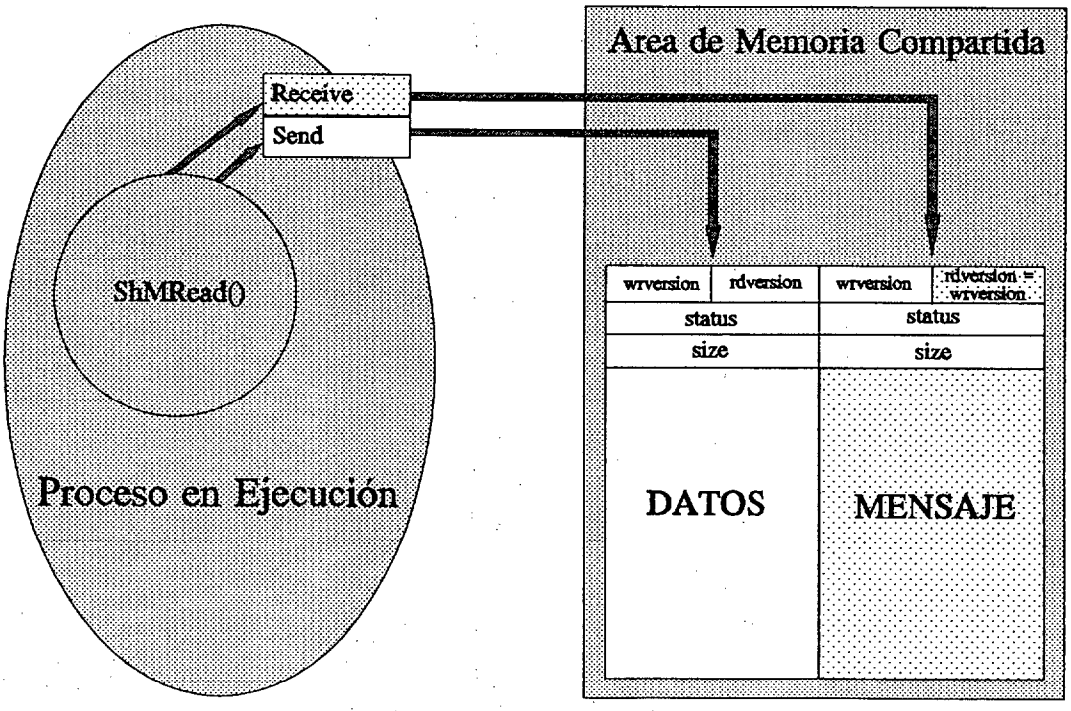

Fig. 5.19 Esquema del proceso de recepción de mensajes a través de memoria compartida.

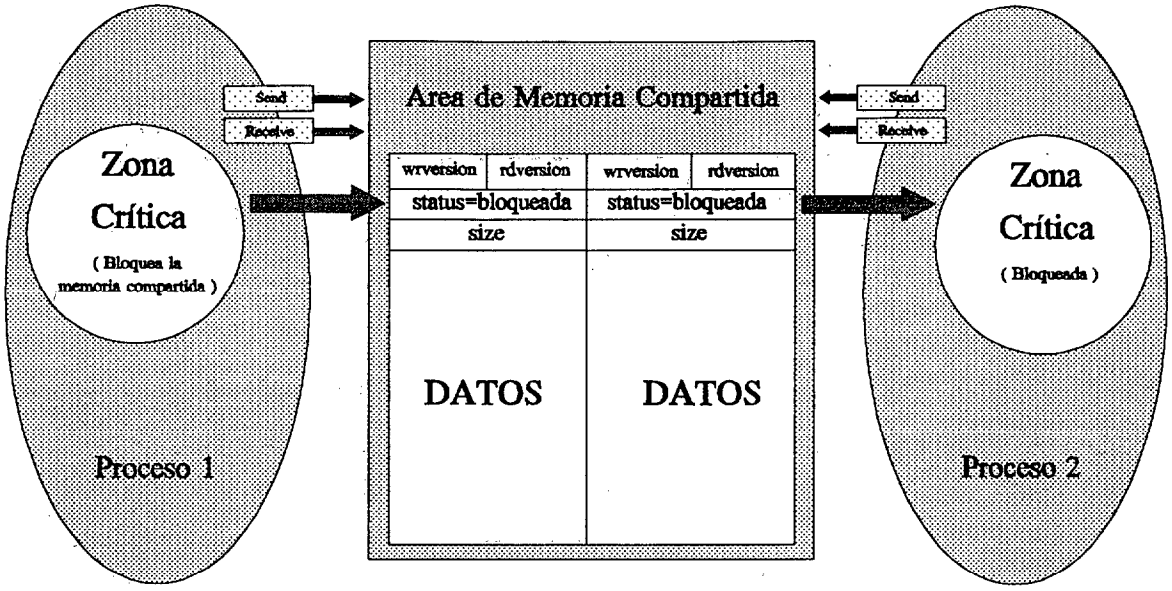

Fig. 5.20 Uso de los bits de estado para generar zonas críticas en los procesos.

5.6.2.3 Gestión de las interrupciones del notificador.

En el entorno gráfico *OpenWindows* el bucle principal de la ejecución del programa, se encuentra en una función del sistema denominada *xv_main_loop()*, como ya se ha visto. Cuando se selecciona algún elemento o se produce algún evento durante la ejecución del programa, la función *xv_main_loop()* llama a la función correspondiente de procesado que se ha definido en el programa. Por lo tanto, si se desea mantener un monitoreo o control constante y reiterativo de algún elemento durante la ejecución del programa nos encontraríamos con la dificultad de que tendríamos que provocar que *xv_main_loop()* llamara a la función de procesado correspondiente. Con los elementos básicos del *OpenWindows* ésto es imposible. Esto es debido, a que si el usuario ejecuta el programa pero no selecciona ningún elemento o provoca ningún evento, la ejecución estaría 'muerta' dentro de *xv_main_loop()*. Las funciones que se hubieran declarado para el procesado constante de datos de entrada que no vengan a través del *OpenWindows* o, que se hubiera declarado para generar salidas constantes fuera del *OpenWindows*, no serían llamadas jamás.

Para prever este tipo de necesidad el *OpenWindows* suministra un paquete de funciones denominado genéricamente **notificador**. Este *notificador* provoca llamadas a funciones definidas ante eventos del sistema (Fig. 5.21). Uno de estos eventos puede ser un temporizador periódico. Con este sistema podemos garantizar que el flujo del programa pasará por ciertas funciones regularmente, aún en el caso de que el usuario no esté seleccionando ningún elemento. De esta forma, podemos mantener un rastreo constante de la entrada de datos. Gracias al notificador podemos garantizar que *SiMon* siempre estará a la escucha de los mensajes que le pueden ser transmitidos a través del área de comunicación.

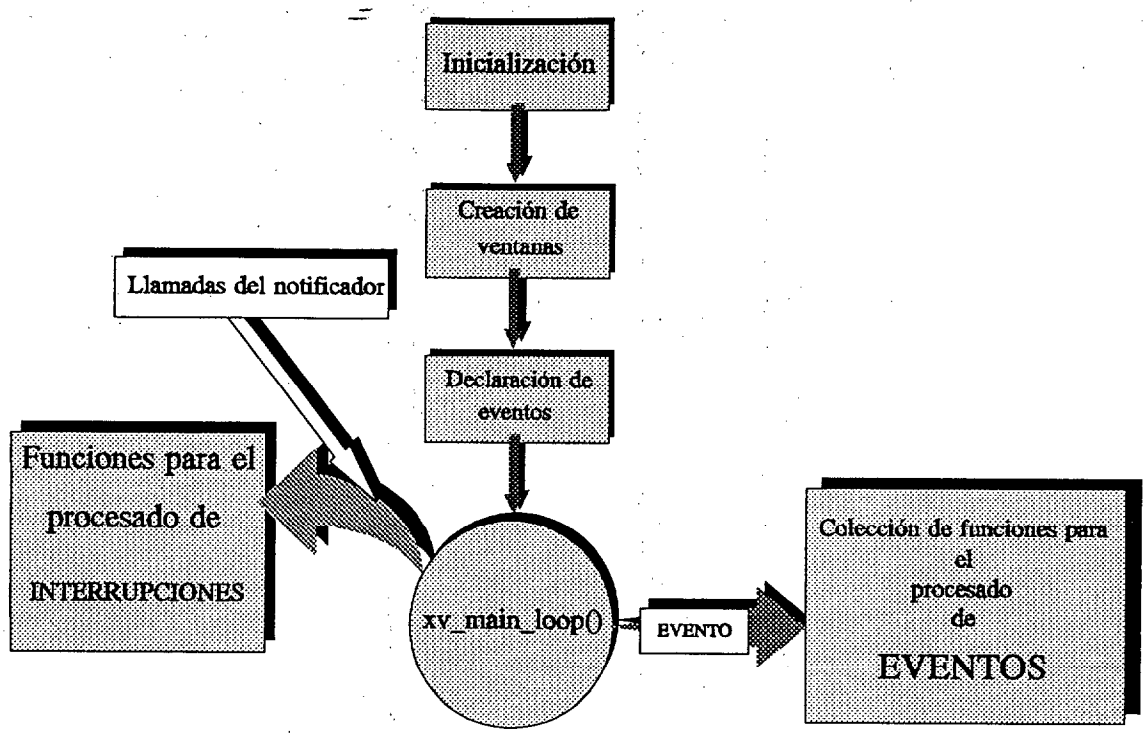

Fig. 5.21 Flujo de control del OpenWindows con notificador.

En la implementación realizada, la cadencia de llamadas a la función de procesado de la entrada de datos, es de diez llamadas por segundo. Esta función de procesado se denomina **Refrescar()**.

5.6.2.4 Descripción del interprete del lenguaje LIV.

El intérprete del *Lenguaje de Intercambio de Variables* puede ser considerado como un autómata cuya ejecución va paralela a la ejecución del programa. El estado de este autómata está relacionado con el valor de la variable **MonStatus**. Los estados de este autómata pueden verse en la Fig. 5.22.

En este gráfico se pueden observar cinco estados. El estado denominado **CLEAR_STATUS** es el de inicialización. Cuando comienza la ejecución de *SiMon* se debe poner el valor de la variable *MonStatus* a ese valor. A partir de este estado vamos al estado de **IDLE**. En este nuevo estado, el intérprete está a la espera de comandos. Los tipos de comandos que pueden venir se pueden agrupar en dos clases. La primera clase son los **comandos de ejecución inmediata**, que se caracterizan porque cuando se van a ejecutar tan sólo necesitan de una instrucción. Este tipo de comandos hace pasar al autómata al estado de **EJECUCION** donde como su nombre indica, se ejecuta la instrucción recibida. Tras ésto, el autómata vuelve de nuevo al estado **IDLE**. El segundo tipo de comandos es aquel que comienza con una instrucción de inicio y a continuación de ésta tiene una lista de parámetros finalizando con un comando **END**. Dentro de este último tipo tenemos dos subtipos diferentes que son **LIST** y **SET**. El comando **LIST** se utiliza para la introducción de sucesivas listas de valores. Cuando aparece un comando de este tipo, el autómata pasa al estado **LIST**. En él, va leyendo todas las líneas como líneas de valores. Estos valores los va ejecutando y los va introduciendo en las variables asociadas a la lista especificada. Si apareciese una instrucción **END** se

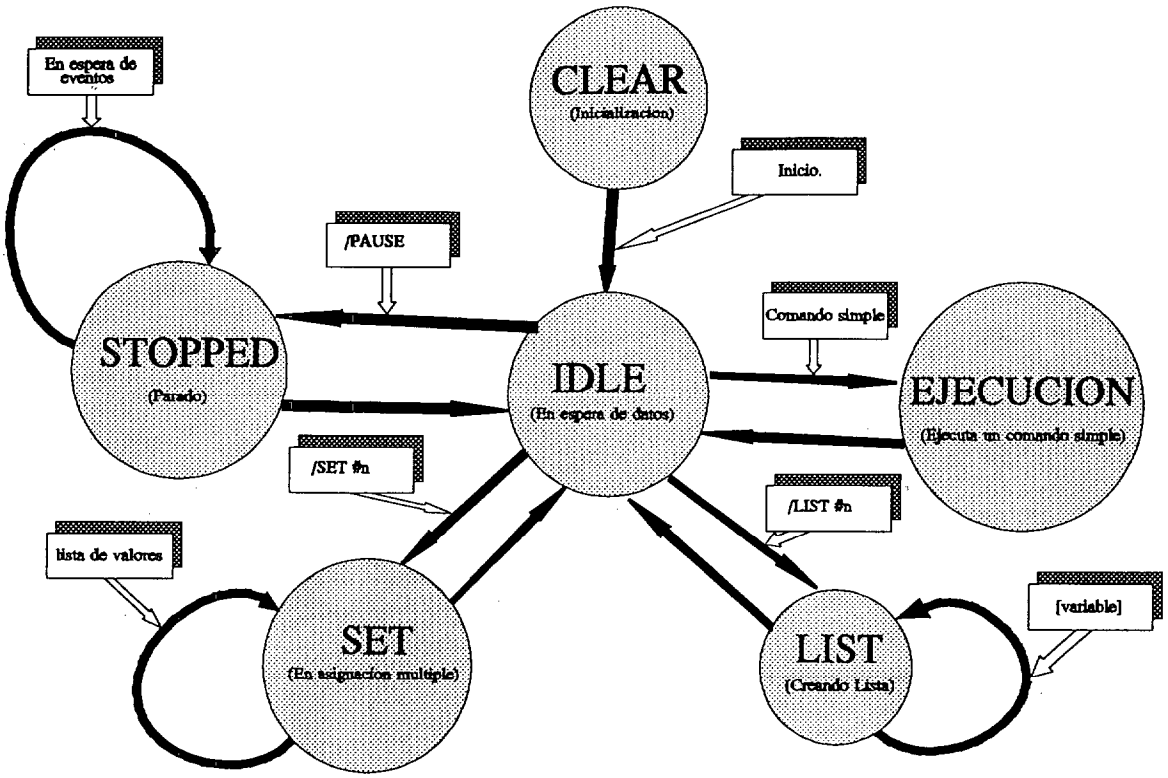

Fig. 5.22 Estados del autómata interprete del LIV.

acabaría la ejecución de este estado pasando el autómata al estado **IDLE**. De igual forma, el comando **SET**, que es utilizado para la introducción de lista de variables, funciona de manera análoga al anterior pero con el estado **SET**.

El estado definido como **STOPPED** es aquel al que pasa el autómata tras el comando **PAUSE**. En este estado se queda a la espera de una entrada, tras la cual pasa al estado **IDLE**.

El punto de entrada para la ejecución del monitor es la función *void Refrescar()*. Esta función lee el área de comunicación de entrada a *SiMon* y ve si contiene datos. Estos datos están formados por texto puesto que, serán instrucciones del *Lenguaje de Intercambio de Variables*. Si no hubiera información disponible, se retornaría, puesto que no habría ningún cambio en el autómata de ejecución. En caso de haber datos, se llamaría a la función ShMonitor(). Esta función lee los datos de entrada como líneas de caracteres. Esta línea es pasada a la función MonSyntax() que se encarga de añadir los datos recién llegados, a una lista de datos de llamadas anteriores que aún no han sido procesadas. Estos datos, por supuesto, son almacenados en variables estáticas. La función *MonSyntax()* devuelve un valor si encontró un comando completo. Se identifica un comando completo, cuando se encuentra un punto y coma. Esto es debido a que el separador utilizado en el lenguaje de intercambio de variables es el punto y coma. De hecho se pueden enviar varios comandos en una misma línea. En *ShMonitor()* se ejecuta la llamada a *MonSyntax()* en un bucle que se sigue ejecutando mientras siga devolviendo comandos. Cuando en los caracteres que queden por procesar no quede un comando completo, *MonSyntax()* devolverá un código vacío, que indicará que han sido procesados todos los caracteres de entrada posibles. Este será el momento en el que *ShMonitor()* concluirá su ejecución.

Tras interpretar cada comando, *ShMonitor()* llamará a la función correspondiente de procesado dependiente del estado actual del autómata.

5.6.2.5 Estructuras utilizadas para el almacenamiento de listas de variables.

Finalmente, en este apartado queremos dejar reflejado como se han almacenado las listas de variables que se utilizan para las asignaciones múltiples. La estructura utilizada tiene el nombre de MonList y tiene la siguiente forma:

```
typedef struct{
        int ID;
        void **list;
        void *link;
}MonList;
```

Esta lista encadenada además del puntero de enlace contiene dos campos adicionales. El primer campo es el identificador del número de lista. Este identificador se utiliza para reconocer una lista en particular. Cuando se declara, se ha de decir a qué lista nos referimos. Así se pueden almacenar varias listas simultáneamente. En la Fig. 5.22, Fig. 5.23 se muestra la forma que tendría esa lista. Todas las operaciones que tengan que ver con estas listas deberán llevar el identificador correspondiente. En casos particulares como el de **CLEAR**, si no llevasen número de lista, se supondría que se refiere a todas las listas por lo que se borrarían todas la listas existentes en memoria. El campo **list**, almacena una lista de punteros a direcciones de variables a las cuales se debe acceder. El último elemento de la lista siempre es un puntero nulo. En el sistema se define una variable global de la forma:

```
MonList *MListas;
```

Esta variable global almacena las listas que han sido declaradas y que el sistema reconoce. Todas las operaciones relacionadas con listas de este tipo siempre toman como raíz esta variable global.

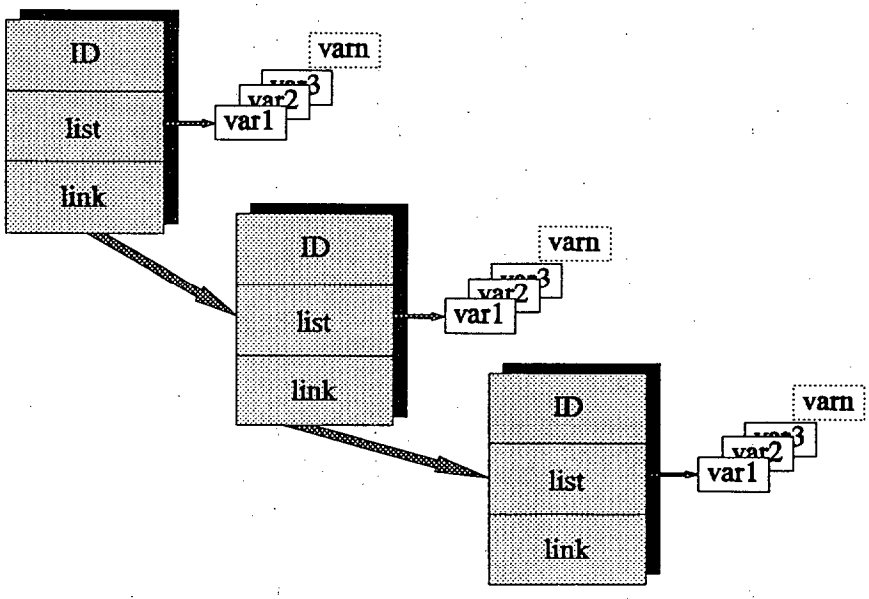

Fig. 5.23 Representación de la estructura que se utiliza para el almacenamiento de listas de variables.

5.6.3 Gestión de las variables de visualización.

En los capítulos anteriores se describieron los formatos de los lenguajes de entrada a *SiMon*. En el se vio que *SiMon* no es solamente un programa de gráficos sino que también tiene la posibilidad de almacenar variables asociadas a elementos dentro del gráfico. En el *Lenguaje de Descripción Visual* se podían declarar variables en cualquier nivel jerárquico. Estas variables han de poder referenciarse desde el exterior. En sí, el *Lenguaje de Intercambio de Variables* podía referenciar a cualquier variable que ya haya sido declarada en el *Lenguaje de Descripción Visual*. Esto nos introduce el problema de cómo identificar cada variable de forma única diferenciándolas de las demás. El problema tiene dos partes que son: cómo generar los nombres únicos para cada una de las variables y cómo se genera la estructura de almacenamiento de los valores de las variables.

A *SiMon* se le podría considerar desde cierta perspectiva como un elemento de almacenamiento de variables. Desde el punto de vista del objeto que este monitorizando y controlando, *SiMon* puede aparecer como tal. En general, el sistema que se monitoriza, suministra valores a variables que van variando por el devenir de acontecimientos sucesivos que van ocurriendo. Este sistema controlado también podría interrogar a *SiMon* por el valor de variables. Por ello, podría ser considerado *SiMon* como una estructura global de almacenamiento de variables.

Al realizar una descripción mediante el *Lenguaje de Descripción Visual*, se utilizan muchos elementos los cuales en sí, no generan variables, ni tienen efecto en la representación gráfica. Nos estamos refiriendo al hecho de que pueden existir declaraciones de elementos, que no sean referenciadas de forma efectiva luego en los diseños. Por ello, no han de tenerse en cuenta a la hora de reservar memoria o generar nombres de variables. Podríamos establecer una analogía con el lenguaje C. En general, en un programa en C podrían existir muchas declaraciones de variables globales o funciones pero, realmente sólo son efectivas aquellas que son referenciadas directa o indirectamente desde la función main() (estas funciones o variables globales se podrían eliminar sin apreciar ningún efecto a la hora de ejecutar el programa). De igual forma, en el lenguaje de descripción visual todos aquellos elementos que no sean referenciados desde un comando *SHEET* no tienen efecto. Por otro lado, un elemento puede ser referenciado varias veces, por eso, el elemento en sí no debe contener elementos para el almacenamiento de variables. Todas las variables las deberá almacenar el *SHEET*. De esta forma, se ha implementado un método para almacenar e identificar de forma simple e inequívoca a cada variable individual.

5.6.3.1 Identificación de variables.

El tema que se va a tratar en este apartado es cómo se va a identificar una variable de forma única. Como ya se comentó en el apartado anterior, una variable sólo tiene declaración efectiva cuando es declarada directa o indirectamente dentro de un *SHEET*. En general, la declaración de una variable tiene sentido dentro de un *SHEET*, un *BLOCK* o un *CELL*.

Variable dentro de un SHEET:

El caso más sencillo es que tengamos declarada alguna variable dentro de un *SHEET*. Podríamos incluir una instrucción *INT* O *PIN* (que son las que declaran variables)

dentro de un *SHEET*. Un método para identificar a esa variable sería poner el nombre con el que son declaradas dentro del *SHEET*. Pero, si utilizamos ese nombre, entonces no podríamos diferenciarla de otra variable que tuviera ese mismo nombre dentro de otro *SHEET*. Por ello, se le ha de poner un prefijo con el nombre del *SHEET* seguido de un punto (Fig. 5.24).

De esta forma, identificamos a cada variable de forma única dentro del *SHEET*, además de saber que pertenece a ese *SHEET* en concreto y no a otro.

```
SHEET SHEJ
   ...
      INT   INT1 ...
      INT   INT2 ...
      PIN   PIN1 ...
      PIN   PIN2 ...
      ...
END

generaría los nombres de variables siguientes:

SHEJ.INT1
SHEJ.INT2
SHEJ.PIN1
SHEJ.PIN2
```

Fig. 5.24 Variable dentro de un SHEET.

Variable dentro de un elemento.

Un caso más general será aquel en el que la variable esté declarada dentro de un elemento que sea referenciado dentro de un *SHEET*. En la Fig. 5.25 se observa que *SHEET* referencia a elementos de nivel jerárquico inferior. En el caso concreto está referenciando al *CELL*, que a parte puede tener una declaración de variables propias. En el ejemplo existen dos referencias a células que tendrán que generar variables independientes, a pesar de estar referenciando al mismo elemento. En realidad, la diferencia entre una y otra referencia son cada uno de los *PLACE*. El nombre de este *PLACE* es el que nos permite identificar de forma única cuando nos referimos a una célula del diseño o a otra.

Aplicando este principio se pueden referenciar elementos complejos desde otros elementos complejos sin limitar el número de niveles jerárquicos. En general, podemos construir un nombre con cuantos elementos hagan falta.

Variables de un ARRAY.

Tan sólo queda un caso especial que es el del ARRAY. La solución de este problema es sencillo. Tan sólo hay que incorporar los índices del ARRAY a continuación del Place que lo referencia (Fig. 5.26).

```
LDV:
   ...
   CELL CELULA
   ...
   INT CELI1
   INT CELI2
   PIN CELP1
   ...
END
   ...

SHEET SHEJ2
   ...
   PLACE P1 ... CELULA
   PLACE P2 ... CELULA
   INT SINT1
   ...
END

VARIABLES QUE SE CREAN:

SHEJ2.P1.CELI1
SHEJ2.P1.CELI2
SHEJ2.P1.CELP1
SHEJ2.P2.CELI1
SHEJ2.P2.CELI2
SHEJ2.P2.CELP1
SHEJ2.SINT1
```

Fig. 5.25 Variable dentro de un elemento.

En resumen, para generar un nombre de una variable hemos de hacer un recorrido *Top-Down*, realizando acotaciones sucesivas del problema. Hemos de identificar primero que nada, a que *SHEET* pertenece la variable. Luego hemos de identificar qué elemento dentro de ese *SHEET* ha generado la variable. Si es un elemento terminal, como un *INT* o un *PIN*, el problema ha sido resuelto. Si es un *PLACE*, se añade el nombre de *PLACE* al nombre completo de la variable. A continuación comprobaríamos qué elemento es referenciado por el *PLACE* e identificaríamos cual es la declaración que hay dentro de ese

elemento que es la que ha generado la variable. En general, hemos de continuar la búsqueda hasta que encontremos un elemento terminal que concluya el nombre. La Fig. 5.27 nos muestra gráficamente el significado de la metodología *Top-Down*.

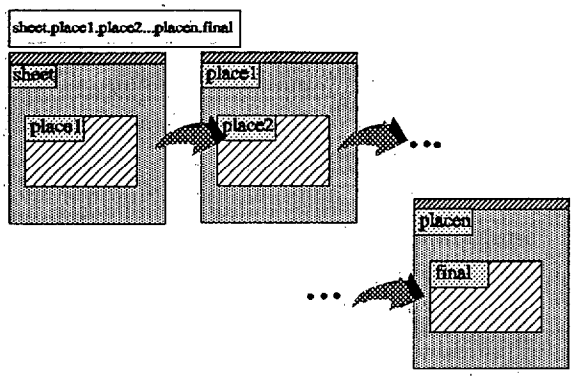

Fig. 5.27 Representación gráfica de la metodología Top-Down.

```
DESCRIPCION LDV

CELL CELULA
        ...
        INT CEL1
        PIN CEL2
        ...
END
        ...
        ARRAY AR ... CELULA
        ...

SHEET SHEJ3
        PLACE P1 ... AR
        PLACE P2 ... CELULA
        INT SH1
END

VARIABLES CREADAS

SHEJ3.P1[0][0].CEL1
SHEJ3.P1[0][0].CEL2
SHEJ3.P1[0][1].CEL1
...
SHEJ3.P2.CEL1
SHEJ3.P2.CEL2
SHEJ3.SH1
...
```

Fig. 5.26 Variables dentro de un ARRAY.

5.6.3.2 Gestión de almacenamiento y recuperación de variables.

En este apartado se va a tratar el tema de cómo poder asignar una dirección de memoria única a cada variable y cómo reconocer cuál es esta dirección. El espacio de almacenamiento de cada variable debe estar reservado de antemano. Por ello, cuando se traduce el código fuente escrito mediante el lenguaje LDV, además de generar la información acerca del sistema, también se han de generar espacios de memoria para generar las variables. Como ya se ha visto, la única declaración efectiva de variables es la que parte de un *SHEET*. Debido a ésto, se deberá reservar memoria para cada *SHEET* y ésta deberá estar asociada a él. Para poder reservar memoria hay que conocer la cantidad que necesitamos. Así, a la hora de procesar el *SHEET*, debemos conocer cual es el tamaño de memoria que ocupan sus variables. Esto se puede calcular a partir de la suma de los espacios de variables necesitados por cada uno de los elementos que contiene el *SHEET*. Este espacio se puede calcular a la hora de declarar cada elemento, de igual forma que se calcula para un *SHEET*. En principio ésto nos introduce en una metodología *Top-Down* para el cálculo de espacios de memoria. Sin embargo, en la implementación se hace siguiendo una metodología *Bottom-Up*.

Cuando se declara algún elemento que contenga variables simples, es decir, instrucciones tipo *INT* o *PIN*, se calcula el espacio de memoria que necesita y se le asocia un indicador de tamaño de espacio de variable a ese elemento. Cuando un elemento de mayor nivel jerárquico referencie elementos de menor nivel jerárquico, los cuales contengan variables, ya se conocerán los tamaños de los espacios de variables de los elementos de menor nivel referenciados. De esta forma, podemos calcular el espacio de memoria necesitado por los elementos de mayor nivel jerárquico. Siguiendo con esta metodología, podemos calcular el tamaño de memoria necesitado por cada *SHEET*. Cuando procesemos la declaración del *SHEET* completa y conozcamos el espacio de memoria necesitado por él, entonces podremos pasar a reservar un área de memoria con el tamaño apropiado que quedará asociado al *SHEET* determinado.

A continuación en la Fig. 5.28 se incluye una declaración a modo de ejemplo para ver el

proceso del cálculo de espacio de variables. Esta declaración es sencilla, dando una idea general del sistema empleado para este cálculo.

Una vez solventado el problema del cálculo de la memoria necesaria para el almacenamiento de variables, hemos de tratar el problema de cómo reconocer la dirección de una variable determinada. De entrada, a cada variable se le ha asignado un espacio único dentro de la memoria reservada. Para la identificación de la posición se ha utilizado una metodología *Top-Down*.

```
...
CELL CELULA ...
...
            INT NOM1 BYTE ...
            #1 BYTE
            INT NOM2 INT32 ...
            #1+4 = 5 BYTES
            ...
END
#TAMAÑO TOTAL DE CELULA: 5 BYTES
ARRAY AR ... 4X4 CELULA
#TAMAÑO DEL ARRAY: 4 FILAS * 4 COLUMNAS * 5 BYTES/CELULA
#TAMAÑO TOTAL DE ARRAY: 80 BYTES
BLOCK B
            ...
            PLACE P1 ... CELULA
            #5 BYTES DE CELULA
            PLACE P2 ... AR
            #5+80 BYTES DE AR = 85 BYTES
            PLACE P3 ... AR
            #85+80 = 165 BYTES
            INT B1 BYTE ...
            #165+1 BYTE DE B1 = 166 BYTES
            ...
END
SHEET SH
            PLACE S1 ... CELULA
            #5 BYTES DE CELULA
            PLACE SB ... B
            #5+166 BYTES DE B = 171 BYTES
            PLACE PR ... AR
            #171+80 BYTES DE AR = 251 BYTES
            PIN PS INT32 ...
            #251+4 BYTES DE PS = 255 BYTES
END
#MEMORIA TOTAL NECESITADA PARA VARIABLES = 255 BYTES
```

Fig. 5.28 Cálculo del espacio ocupado por variables.

5.6.4 Interfaz gráfico de Simon.

5.6.4.1 Transformaciones gráficas.

Antes de entrar a describir cómo se dibujan los elementos del diseño y cómo se procesa la información gráfica contenida en las descripciones hemos de tratar el tema de las transformaciones gráficas.

Si nos fijamos en la información gráfica contenida en los sistemas que pueden ser descritos por el lenguaje de descripción vemos que éstos contienen primitivas gráficas que son utilizadas en elementos más complejos para formar bloques. Un bloque puede a su vez, formar parte de otro bloque o diseño. De tal suerte, que existen infinidades de niveles jerárquicos que podemos definir. En la práctica, la limitación para declarar diferentes niveles jerárquicos no es más que la memoria del ordenador que estemos usando.

Todos los comandos gráficos insertados en cualquier primitiva, ya sea una célula (*CELL*), un bloque (*BLOCK*), o un diseño (*SHEET*), están referidos al origen del elemento en el que están insertados. Cuando decimos que hemos de dibujar una caja, una línea, un punto, escribir un texto, o cualquier otro tipo de información gráfica en una célula, siempre nos estaremos refiriendo al origen de la célula. Esto se muestra en la Fig. 5.29. Este origen no es absoluto, ya que será relativo al origen en que se ha posicionado esta célula. Esta célula podría estar posicionada dentro de un bloque. El punto en que se posiciona la célula dentro del bloque será el origen para todas las primitivas gráficas de la célula. Al posicionarse una célula, ésta puede estar rotada o estar sometida a factores de escala. Estos parámetros afectan a todos los elementos gráficos de la célula. Todo lo dicho anteriormente para una célula se puede aplicar a un bloque puesto que éste podría ir dentro de otro bloque. Por lo tanto, habríamos de aplicar

todas las transformaciones gráficas de nuevo. Los posicionamientos, como ya se ha dicho, no son absolutos, excepto en los *SHEET*. En principio, en un *SHEET*, un posicionamiento se refiere a un punto absoluto dentro de la pantalla. Sin embargo, ni siquiera ésto es cierto. Se ha de tener en cuenta que un *SHEET* puede tener un escalado hecho por culpa de un zoom que ha hecho el usuario para apreciar algún detalle. Además el usuario puede haber desplazado el origen del *SHEET* para que en la pantalla gráfica quede una célula en concreto. Junto a la información gráfica contenida en la descripción, existe otra información gráfica que define el usuario en tiempo de ejecución y que es totalmente arbitraria.

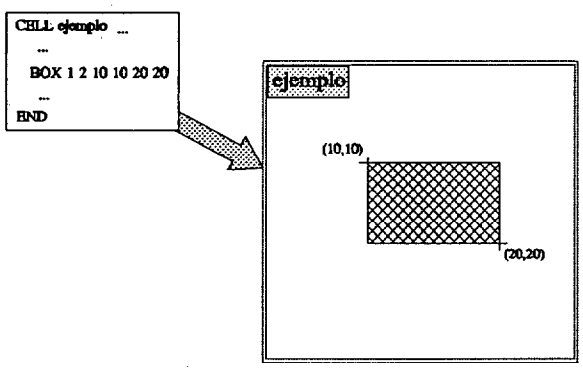

Fig. 5.29 Representación de una caja dentro de una célula.

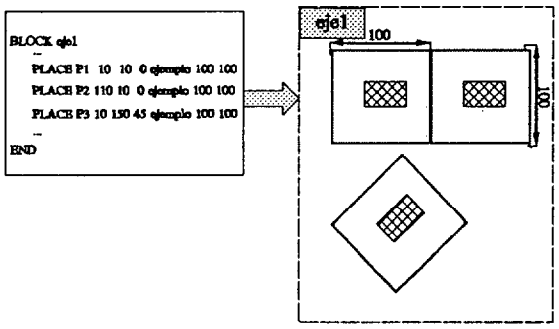

Fig. 5.30 Representación de una célula dentro de un bloque.

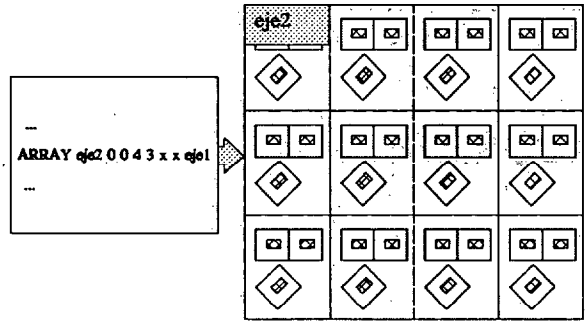

Fig. 5.31 Réplica de un bloque dentro de un array.

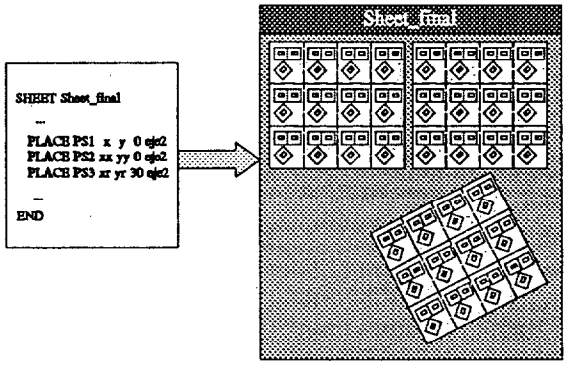

Fig. 5.32 Representación del array dentro del sheet final.

El hecho de que las coordenadas del comando gráfico sean relativas al ambiente en que estos comandos son declarados implica que hemos de aplicar una lista de transformaciones a estas coordenadas. Esta lista de transformaciones estará relacionada con los parámetros de dibujo del ambiente en el que están insertadas. Tal como se comentó, si una célula está posicionada en un punto dentro de un *SHEET*, todos los elementos de dibujo que se encuentran dentro de dicha célula tendrán el desplazamiento de la célula. Además si está escalada o rotada esa célula, todos los comandos gráficos estarán escalados o rotados en la medida que lo esté esa célula. Si en vez de estar insertada en un *SHEET*, estuviera en un bloque, el *SHEET* sería el mismo, es decir, si el bloque estuviera posicionado dentro de otro bloque o *SHEET*, tendríamos que transformar todas las coordenadas de los comandos gráficos que genere ese bloque. Para realizar este proceso se ha implementado una lista de transformaciones gráficas. Un nodo de esta lista está declarado de la forma siguiente:

```
typedef struct {
        int command;
        int x;
        int y;
        int misc;
        void *link;
} CmList;
```

Un esquema del funcionamiento de este nodo se muestra en la Fig. 5.33. El campo command de la lista contiene el tipo de transformación que contiene la lista. Se puede definir cualquier tipo de comando. En la implementación del interfase gráfico se han declarado los siguientes comandos:

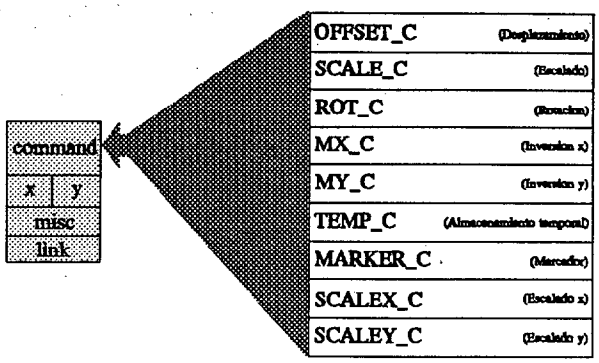

Fig. 5.33 Nodo básico de la lista de transformaciones gráficas.

- **OFFSET_C**
- **SCALE_C**
- **ROT_C**
- **MX_C**
- **MY_C**
- **SCALEX_C**
- **SCALEY_C**

Cuando se está procesando un gráfico para representarlo en pantalla, todos los comandos PLACE generan varios nodos con las transformaciones que hay que hacer para posicionar el objeto al que se refiere ese PLACE. Así, la lista encadenada va creciendo con todos los nodos que van insertando los PLACEs que nos vamos encontrando. Finalmente, cuando aparecen órdenes de dibujo reales tales como: líneas, puntos, cuadros ... hemos de transformar cada una de sus coordenadas independientemente utilizando la información contenida en la lista de transformaciones. En la práctica, aparte de los comandos PLACE también generan nodos de información los comandos *ARRAY*s. El comando *ARRAY* genera un nodo que va variando para cada uno de los elementos que hay que dibujar. Como se verá la función de dibujo de los *ARRAY*s toma el control una vez por cada uno de los elementos que dibuja y varía el nodo que ha insertado en la lista para obtener la lista correcta para cada elemento.

El comando **OFFSET_C** (desplazamiento) genera un desplazamiento de la coordenada que se procese. Es decir, a las coordenadas x e y que le entren a la función de procesado se le habrá de sumar las coordenadas x e y contenidas en el nodo de transformación.

El comando **ROT_C** (rotación) implica que la coordenada que se está procesando ha de ser rotada con respecto al punto (0,0) *misc* grados (*misc* es una variable asociada al nodo de transformación).

El comando **SCALE_C** (escalado) provoca un escalado de la coordenada en proceso, es decir, los valores x e y de las coordenadas se multiplican por un factor. Este factor es un número racional donde su numerador sería el parámetro x del nodo y su denominador sería el parámetro y del nodo.

Los comandos **MX_C** y **MY_C** (inversión) son comandos *mirror*. **MX_C** invierte el parámetro x de la coordenada que se esté procesando. De igual forma el comando **MY_C** invierte el parámetro y de la coordenada en proceso.

Los comandos **SCALEX_C** y **SCALEY_C** (escalado selectivo) escalan un sólo eje. **SCALEX_C** funciona igual que **SCALE_C** pero tan sólo para el parámetro x de la coordenada. **SCALEY_C** funciona igual que **SCALE_C** pero sólo para la coordenada y.

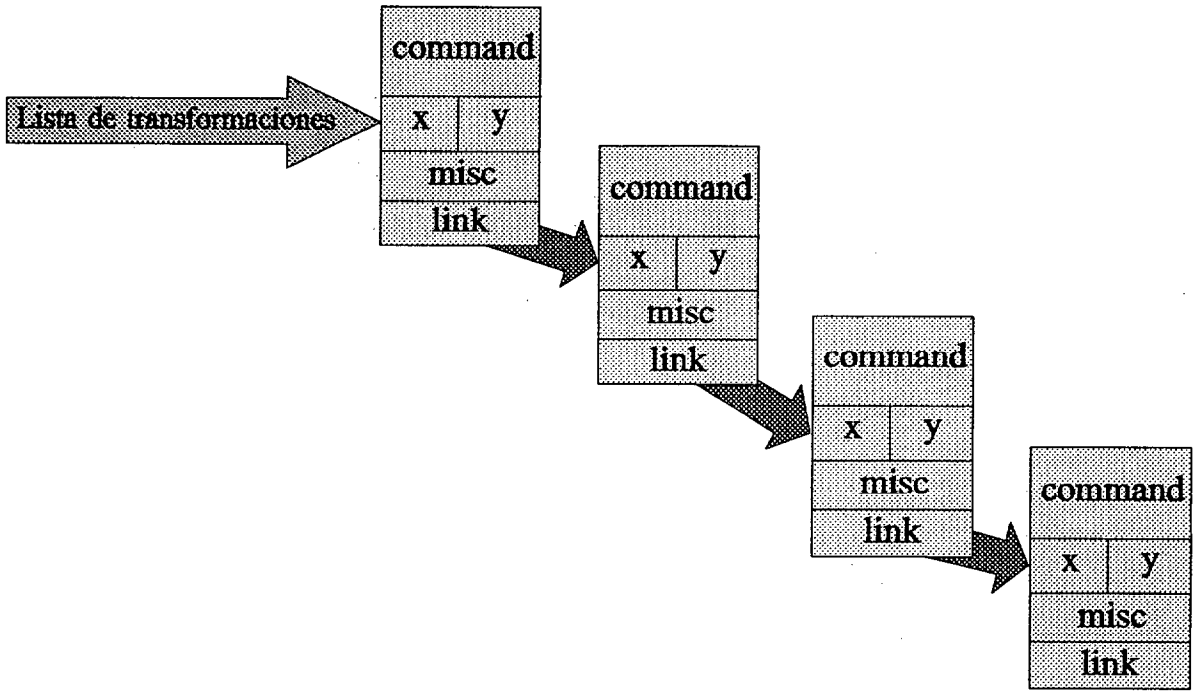

Fig. 5.34 Lista de transformaciones gráficas.

El parámetro *link* de la lista de transformaciones es el puntero que se utiliza para formar la lista encadenada de transformaciones. La lista completa de transformaciones queda tal como se muestra en la Fig. 5.34. En ella aparecen una sucesión de nodos, los cuales están ordenados según la secuencia en la que fueron introducidos. Esto, como se ha visto, es importante tenerlo presente a la hora de procesar la información contenida en la lista.

Se han implementado funciones para la inserción, borrado, y proceso de esta lista de transformaciones. Siempre que se inserte un nodo, éste quedará como cabeza de lista. La función de inserción de nodos en la lista tiene el siguiente formato:

CmList *Inserta(CmList *lista, CmList *info)

A esta función se le pasa la lista en la que se desea insertar y el nodo que se desea insertar como parámetros y devuelve un puntero a la cabeza de lista. Así mismo, para borrar un nodo existe una función propia llamada *Delete* con el siguiente formato:

CmList *Delete(CmList *lista, CmList *info)

A la función *Delete* se le pasa como primer parámetro la lista de la que se desea borrar un nodo y un puntero al nodo que se desea borrar. Esta función devuelve un puntero a la cabeza de lista. Hemos de tener en cuenta que hay que tomar como cabeza de lista el puntero devuelto por ambas funciones. Esto es debido a que estas funciones pueden cambiar la cabeza de lista (no es seguro que un nodo pueda ser insertado por tanto no sabremos cual es la cabeza de lista si no tomamos dicho puntero). De igual forma podemos tratar de borrar la cabeza de lista y ésta variaría.

A parte de estas dos funciones de gestión de la lista de transformaciones se han implementado otras de procesado de las transformaciones implícitas en la lista.

void Transforma(CmList *lista, int *x, int *y)

A esta función se le pasa como parámetros la lista de transformaciones que vamos a utilizar y los valores *x* e *y* de la coordenada. Nótese que los valores *x* e *y* de la coordenada se pasan por dirección porque la función *Transforma* varía estos parámetros lo cual hace inútil el pasarlos por valor. Esta función va cogiendo todos los nodos de la lista, desde la cabeza a la cola, y aplica las transformaciones implícitas en cada nodo. Además existe otra función de transformación que realiza la transformación inversa a la contenida en una lista. La declaración de esta función es la que sigue:

void InvTransform(CmList *lista, int *x, int *y);

Esta función es muy útil para conocer el valor relativo de una coordenada absoluta según la lista de transformaciones. Esta función se utilizará para identificar las células individuales picando en una representación gráfica.

Con esta estructura y estas funciones el problema de las transformaciones gráficas se ha reducido a introducir nodos de transformación gráfica en una lista.

5.6.4.2 Dibujo de las descripciones.

Una vez descritos los elementos que intervienen en el dibujo de descripciones, pasamos a continuación a describir como se han implementado las funciones para realizar tal tarea.

Con la descripción del sistema ya procesada y en memoria y con la memoria para almacenamiento de las variables generadas podemos comenzar a dibujar el sistema. Para ello, debemos tener en cuenta ciertos aspectos:

1) por cada *SHEET* descrito se generará una ventana con representación gráfica y con todos sus elementos de selección.

2) sólo se dibujarán aquellos elementos que sean referenciados en un *SHEET*. Esto significa, que cualquier declaración que se haga no tiene necesariamente que generar una representación (tan sólo generará una representación si es referenciada en un *SHEET*).

3) serán tenidas en cuenta todas las transformaciones gráficas que van generando todos los elementos de la descripción que se van procesando.

Se ha de tener en cuenta que tan sólo han de ser procesadas aquellas instrucciones que generan alguna salida gráfica. Estas instrucciones son:

PLACE.

Esta instrucción nos dice que situemos algún elemento en la representación. El elemento que referencia tiene que ser un elemento complejo que lleve alguna

declaración o referencia a otro elemento. Concretamente un PLACE sólo puede referenciar a *BLOCK*, *ARRAYS*, *CELL*. El comando PLACE nos indica en que posición respecto al origen relativo del elemento en el que está declarado hemos de situar el elemento referenciado, así como la rotación. Opcionalmente puede llevar parámetros de *SCALE_X* y *SCALE_Y*. El comando PLACE sólo genera varios nodos de transformaciones que se añaden a la lista y provoca una llamada que procesa el elemento de dibujo que referencia. Cuando esa llamada retorna han de ser borrados los nodos de la lista que se generaron puesto que sólo pertenecen a ese PLACE.

TEXT,LABEL.

Estos comandos generan instrucciones de dibujo efectiva. Son instrucciones para generación de texto y variables. En los comandos ya van implícitas las coordenadas de dibujo. Por supuesto, se han de tener en cuenta la lista de transformaciones gráficas existentes.

INSTRUCCIONES GRAFICAS ELEMENTALES.

Nos referimos al dibujo de primitivas gráficas tales como líneas, puntos, rectángulos... Cuando aparecen estas instrucciones generan comandos gráficos efectivos. Al igual que las instrucciones del apartado anterior se han de transformar sus coordenadas para respetar la lista de transformaciones existente.

BLOCK, ARRAY, CELL.

Las declaraciones contenidas en la descripción de estos elementos tan sólo generan representación gráfica cuando son referenciadas por un PLACE de un *SHEET* o un PLACE de algún elemento que sea referenciado directamente o indirectamente desde un PLACE. Para procesar la información contenida en la descripción de estos elementos se ha de realizar un recorrido por ella y generar los comandos de procesado gráfico para todas las instrucciones que puedan generar salida gráfica.

La instrucción *ARRAY* si bien se ha incluido en este apartado es un poco diferente. Esta instrucción genera múltiples llamadas a la función de procesado del elemento que referencia. Para cada llamada genera un nodo de transformación que contiene el offset relativo de ese elemento y que está relacionado con sus índices dentro del *ARRAY* y con los valores de incremento que se especifica en la instrucción *ARRAY*.

En la Fig. 5.35 se muestra un ejemplo, de como se produce la cadena de llamadas para dibujar la representación gráfica de un sistema. En ella se muestra como se va descendiendo en la descripción a través de la estructura de datos que almacena la descripción.

En el esquema se representa a cada llamada a una función como un nodo. Existe recursividad que generalmente es indirecta. Es decir, normalmente no se da el caso de que una función se llame a sí misma y sí es muy común que una función sea llamada por otra función que ha sido llamada directa o indirectamente por la primera función. Esto está hecho así porque no

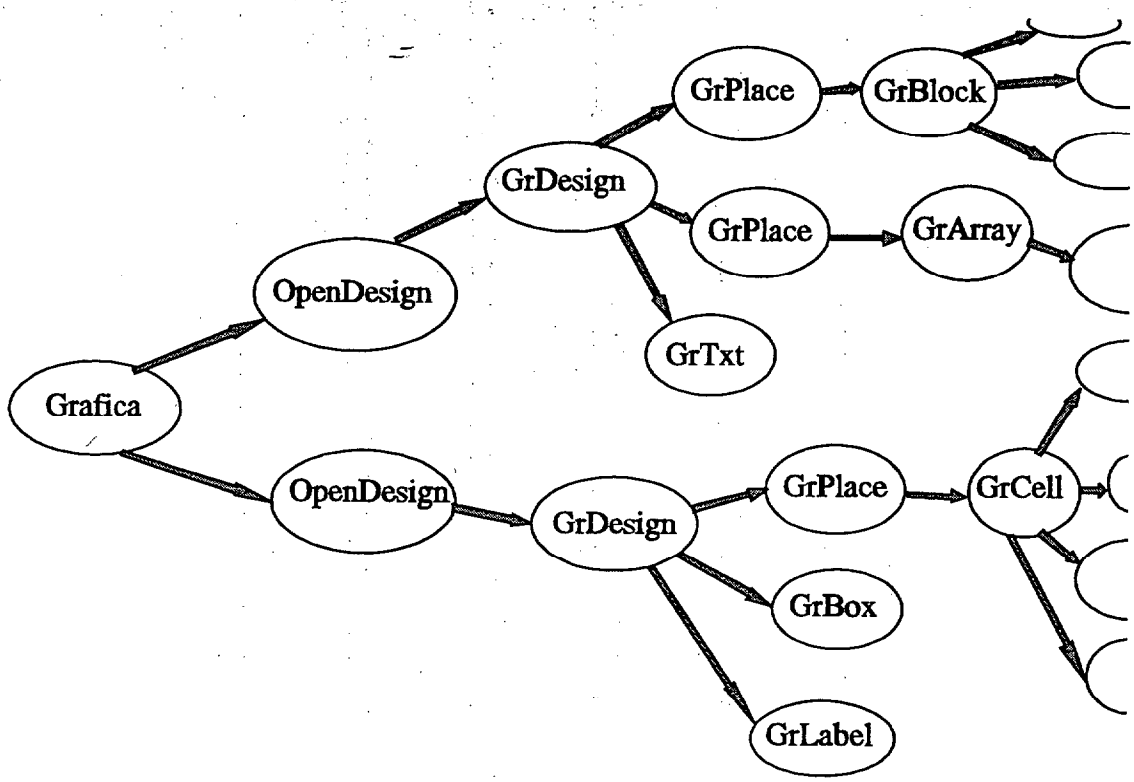

Fig. 5.35 Ejemplo de la cadena de llamadas generada para dibujar la representación de un sistema.

existe limitación en la jerarquía de la descripción. Ya entrando en detalle la utilidad de cada función es:

int Grafica(Generic *n);

A esta función se le pasa la raíz de una descripción completa. Esta función crea las ventanas gráficas, genera los colores y deja listo el sistema para la atención a las ventanas creadas. En la práctica esta función tras inicializar los colores recorre la descripción buscando comandos *SHEET* y llama a la función **OpenDesign**() con la dirección de la descripción de cada *SHEET* encontrado.

void OpenDesign(BloDes *Des);

Esta función crea la ventana asociada a un *SHEET*. Se crea una ventana completa con su subventana gráfica, sus paneles con sus elementos de selección y su cabecera. Esta ventana permanece invisible hasta que el usuario a través del interfase gráfico decide activarla. A parte de ésto esta función asocia al *SHEET* una lista de transformaciones nulas. Esto es debido, a que los parámetros de ZOOM y desplazamientos a los que pueda estar sometido el dibujo son almacenados por cada *SHEET*. Se deduce que en cada ventana se tendrá un ZOOM y un desplazamiento diferente. Inicialmente no habrá ningún desplazamiento ni ningún ZOOM. Además, los parámetros adicionales de los *SHEET* que mantienen los parámetros gráficos asociados a éste son inicializados. En la práctica estos parámetros son: el identificador de *Frame*, el identificador de *Canvas*, el identificador de *Panel*, las ventanas de parámetros asociados a esta ventana y el tipo de ZOOM que se hará por defecto. En la práctica esta función no llama a ninguna otra función para que

realice el dibujo contenido en el *Canvas*. Esto es debido, a que el *OpenWindows* automáticamente llamará a la función de repintado asociada al *Canvas* correspondiente. Por ello, el dibujo del sistema termina aparentemente aquí. En la práctica hemos de considerar que al crear el *Canvas* serán llamadas las funciones de repintado asociadas. Por ello, si bien no existe flujo de control directo desde esta función, seguiremos contando las funciones del esquema como si fueran a continuación de ésta.

int GrDesign(Canvas Cv, Xv_window pw, Display *dpy, Window xwin, Xv_xrectlist area);

Esta función es llamada cuando se crea el *Canvas* por el **OpenDesign()** y cuando algún elemento gráfico varía. En general, el sistema llamará a esta función cada vez que detecte que se necesita un repintado de la pantalla gráfica. Esto puede ser motivado porque el usuario haya cambiado el tamaño de la ventana gráfica, ha movido el sistema, ha cerrado el icono, ha abierto el icono... También podemos llamar a esta función por necesidades propias para reflejar variaciones en los parámetros, variaciones de elementos gráficos, alteraciones en las visibilidades de los niveles... El conjunto de parámetros que se le pasa a la función está definida y fijada de antemano por la herramienta *OpenWindows* que cuando llama a la función lo hace con estos parámetros. Realmente cuando llamamos a la función no podemos conocer todos estos parámetros o si lo intentáramos conocer llevaría una cantidad de proceso importante. Por ello, para optimizar el proceso se ha almacenado en *SHEET* todos lo parámetros gráficos necesarios para poder dibujar en la ventana gráfica. El único parámetro importante es el que nos identifica el *Canvas* que genera la llamada a la función repintado (esto nos sirve para identificar qué *SHEET* ha generado la llamada). Puesto que la raíz de la descripción es una variable global, lo primero que nos hace la función GrDesign es recorrer la descripción del sistema buscando qué *SHEET* tiene asociado el *Canvas* que se le ha pasado como parámetro. Si identifica algún *SHEET* como propietario de ese *Canvas*, genera la función de repintado que lleva implícito ese *SHEET*. No es posible que se genere una llamada a una función de repintado con un *Canvas* que no esté en ningún *SHEET*. No obstante, previniendo fallos del sistema se retorna sin hacer nada en caso de no encontrar el *SHEET*. Una vez identificado el *SHEET* que ha generado la llamada a la función de repintado se pasa a generar los comandos de dibujo, interpretando la descripción del *SHEET*. En la práctica, lo que se hace es un recorrido total por los nodos de información asociados a la descripción del *SHEET*. Se ignoran los nodos que no contengan información gráfica y se procesa los que sí tienen. En un *SHEET* sólo generan información gráfica los comandos de dibujo simple de líneas, rectas .., y los comandos PLACE. Se pasa como lista de transformaciones aquella que lleva asociada el *SHEET*. Hay que tener en cuenta que esta lista si bien al principio se pone como lista vacía, no siempre lo estará. Esto es debido, a que la operación del usuario sobre la ventana puede provocar desplazamientos, ZOOM, y otros efectos que se almacenan en la lista de transformaciones asociada al *SHEET*. En la práctica, el recorrido de la descripción se hace tantas veces como niveles existan (recorriendo los niveles más bajos primero). De esta forma, podemos decidir que elementos se superponen en el *SHEET*. Las funciones que generan instrucciones de dibujo

efectivas para el *OpenWindows*, no hacen nada si el nivel en el que se está pintando en el momento en que se les llama no coincide con el nivel asociado al elemento que se trata de pintar.

void GrPlace(BloDes *d, Place *Nodo, CmList *list, byte level);

Esta función es llamada por todos los elementos que puedan contener una función PLACE (en la práctica sólo son *SHEET* y *BLOCK*). Se le pasa como parámetro el *SHEET* original del que parte la cadena de llamadas de repintado hasta este PLACE que se pretende procesar y el nivel del dibujo actual. Esto es debido, a que los parámetros gráficos están almacenados en el nodo del *SHEET*. **GrPlace()** inserta un nodo de transformación de desplazamiento en la lista de transformaciones, otro nodo de rotación y opcionalmente dependiendo de si en la declaración del PLACE en curso se ha puesto o no escalados selectivos, uno o dos nodos de escalado (horizontal o vertical). Tras ésto, identifica el tipo de elemento que esta referenciando. Basándose en este tipo genera la llamada correspondiente. Luego, borra de la lista de transformaciones los nodos que insertó previamente.

void GrBlock(BloDes *d, BloDes *Nodo, CmList *list, byte level);

Esta función es muy similar a GrDesign. En la práctica, un *BLOCK* es como un *SHEET* pero que no genera una ventana. Se le pasa los mismos parámetros que a **GrPlace()** y al resto de las funciones. El proceso de *BLOCK* es igual al de *SHEET* sólo que se hace un recorrido único con el nivel actual. Otra característica que tiene es que no genera nodos para la lista de transformaciones y sí genera una sucesión de llamadas por cada una de los PLACE o instrucciones efectivas de dibujo que se encuentran en su descripción.

void GrArray(BloDes *d, Array *Nodo, CmList *list, byte level);

Esta función desde el punto de vista de la representación es muy similar a **GrPlace()** salvo que ésta genera una sucesión de llamadas al elemento de descripción que contiene. La diferencia que existe entre cada llamada es el nodo de transformación que se inserta en la lista. Al inicio, esta función inserta un nodo de transformación de desplazamiento en la lista de transformaciones gráficas. El valor que contiene este nodo inicialmente es el valor de desplazamiento inicial que se encuentra en la declaración de *ARRAY*. Tras el retorno de cada llamada, esta función actualiza el nodo insertado para que cada llamada tenga una consecuencia gráfica diferente (de esta forma, barremos todos los elementos del *ARRAY*). Por definición los elementos que puede contener un *ARRAY* sólo son *CELL* o *BLOCK*. Cuando se han procesado todos los elementos del *ARRAY*, esta función borra el nodo que insertó en la lista de transformaciones para dejarlo tal cual estaba cuando fue llamada.

void GrCell(BloDes *d, Cell *Nodo, CmList *list, byte level);

El parámetro *Nodo* al contener una descripción del mismo tipo que *BLOCK* y *SHEET* tiene un procesado muy similar. En la práctica, lo que hace es recorrer

su descripción y generar una llamada por cada instrucción que tenga significado gráfico efectivo. Téngase en cuenta que esta instrucción no puede contener ningún PLACE.

void GrBox(BloDes *d, Box *Nodo, CmList *list, byte level);
void GrLinRec(BloDes *d, LinRec *Nodo, CmList *list, byte level);
void GrPol(BloDes *d, Pol *Nodo, CmList *list, byte level);

Estas tres instrucciones son instrucciones gráficas efectivas. Esto es así, porque estas instrucciones siempre generan llamadas gráficas efectivas al entorno *OpenWindows*. Pudiera ser que una *CELL*, *BLOCK* o *SHEET* al ser dibujada, genere una sucesión de llamadas a las diferentes funciones que estamos comentando,pero que al final no se generara ninguna instrucción efectiva de dibujo. No es obligatorio que un *SHEET* contenga elementos gráficos. Esto puede parecer ilógico, sin embargo, debemos admitir que el usuario no está obligado a representar nada, por lo tanto, en un *SHEET* vacío en cuanto a instrucciones gráficas no se generarán llamadas a funciones de dibujo del *OpenWindows*, sin bien, al procesar la descripción, se generará una sucesión de llamadas tal como la que estamos estudiando. La función **GrBox()** dibuja una caja tal como la que se indica en el nodo. Esta función cuando es llamada ve si el nivel que se le pasa corresponde con el nivel de dibujo. Si es así, procesa las coordenadas que contiene según la lista de coordenadas que se le ha pasado y genera la caja. En la práctica, la caja es generada en dos pasadas. Cuando el nivel que se le pasa en level coincida con el nivel correspondiente al fondo o al rellenado, generará uno u otro. La función **GrLinRec()** genera una línea o rectángulo según el tipo de nodo que se le pase. La función **GrPol()** dibuja un polígono relleno según los datos que se le pasan en el parámetro Nodo. Esta función es muy similar a **GrBox()**, sólo que en vez de tener dos coordenadas, que serían los extremos del rectángulo, aquí se le pasan los vértices de un polígono. Cuando estas funciones son llamadas es seguro que van a dibujar.

void GrTxt(BloDes *d, Txt *Nodo, CmList *list, byte level);
void GrLabel(BloDes *d, Label *Nodo, CmList *list, byte level);

Estas dos funciones también son instrucciones efectivas de dibujo. Estas funciones dibujan texto en la representación. Este texto puede estar escalado. Respeta la lista de transformaciones gráficas. Además **GrLabel()** escribirá el valor de la variable a la que esta representando.

Aparte del funcionamiento normal existe un modo especial que puede estar indicado por el bit denominado **FullRf**. Este bit se encuentra en la estructura de parámetros gráficos que tiene asociado cada *SHEET*. Si este bit estuviera a falso, ninguna de las funciones anteriores dibujarían nada. Tan solo generarían dibujo las funciones **GrLabel()** cuya variable asociada haya cambiado de valor. Este efecto se utiliza para obtener un refresco dinámico de los valores de las variables, sin tener que repintar el gráfico completo. Esto libera al sistema de una gran cantidad de proceso innecesario, y da mayor agilidad.

En general **GrDesign()**, es el origen del proceso de dibujado, de la representación. Cuando se da la orden de redibujar se cambia un parámetro, un valor, el control de visibilidad varía

los niveles visibles ... Siempre se llama a la función **GrDesign()** con el *Canvas* sobre el que se pretende dibujar. **GrDesign()** reconoce el *SHEET* destino por el *Canvas*. Cuando se llama a la función **GrDesign()**, se desencadena un proceso en el cual es llamada, una sucesión de funciones dependiendo de la descripción de **GrDesign()** y de cada elemento perteneciente al *SHEET* que se pretende dibujar. Existe recursividad indirecta puesto que, un *BLOCK* puede estar dentro de la descripción de otro *BLOCK* a través de un PLACE. En la práctica, no está garantizado que esta sucesión de llamadas genere un gráfico. Esto es debido, como ya se comentó, al hecho de que la descripción puede estar vacía de contenido gráfico. Lo que sí es cierto es que se recorren todos y cada uno de los elementos de la descripción buscando significados gráficos en ellos. Se ha de tener claro que **GrDesign()** provoca un número de llamadas igual al número de niveles activos existentes multiplicado por el número de elementos con posible significado gráfico que se encuentre.

Cada *SHEET* contiene una lista de niveles activos, los cuales nos dicen los niveles visibles en el gráfico. Por ello el estado de los niveles activo o desactivado, es independiente para cada *SHEET*.

5.6.4.3 Transformaciones dinámicas en tiempo de ejecución.

Una vez realizado la representación del sistema, el usuario puede realizar diferentes transformaciones en tiempo de ejecución. Básicamente estas transformaciones se refieren a desplazamiento del esquema, zoom o selección de elementos individuales dentro del esquema representado. Estas transformaciones son básicas en cualquier entorno interactivo y su funcionalidad es destacar o modificar alguna variable del sistema. Para realizar esta tarea se utilizan unas facilidades proporcionadas por el entorno *OpenWindows* denominadas eventos.

Al definir la ventana principal del entorno, se le indica al sistema que debe notificar los eventos de pulsar los botones del ratón a cierta función predefinida. Así, las únicas entradas al programa en tiempo de ejecución son la notificación de eventos y la pulsación de botones definidos en paneles.

Las acciones que puede realizar el usuario en tiempo de ejecución se pueden dividir en tres grupos:

- **Transformaciones gráficas.**
- **Selección de elementos del diseño.**
- **Control del entorno.**

5.6.4.3.1 Transformaciones gráficas.

En este apartado se van a comentar las funciones que realizan el *zoom* y el desplazamiento del esquema representado. Estas opciones nos dan la posibilidad de ver una parte del diseño ampliada, seguir la traza de funcionamiento de un elemento determinado y poder representar el diseño en cualquier posición y forma que nos sea más útil para la comprensión de los datos que está representando.

Para la realización de estas transformaciones se utiliza la lista de transformaciones gráficas. En la práctica, cuando se ha interpretado la orden del usuario, lo que se hace es introducir

nodos de transformaciones gráficas al diseño, que nos provoquen el efecto deseado por dicho usuario.

Téngase en cuenta, que los nodos tipo SHEET contienen un puntero a una lista de transformaciones. Esta lista nos mantiene la transformación seleccionada por el usuario y así mismo, nos permite mantener una transformación independiente para cada SHEET. En la práctica, tras insertar un nodo en la lista se realiza un postprocesado de ésta. Esto es debido, a que una sucesión de desplazamientos y escalados pueden ser transformados en un único desplazamiento global y un único escalado global. Es muy importante realizar esta conversión puesto que cuantos más nodos existan en la lista mayor tiempo de CPU se consumirá y por otro lado, los errores de cálculo aumentarán.

a) Desplazamiento.

El desplazamiento se refiere a la posibilidad de mover el gráfico a cualquier punto de la ventana. Esto se puede realizar de dos formas: **fijo** y a **medida**.

El ***desplazamiento fijo*** se realiza seleccionando uno de los botones disponibles en el panel de la ventana. Existen cuatro botones, los cuales nos permiten desplazar el diseño en dirección hacia los cuatro puntos cardinales.

En la práctica, lo que se hace es introducir un nodo de desplazamiento en la lista de transformaciones gráficas, que almacenará un vector de desplazamiento dependiendo de la dirección en la que se desee desplazar. El módulo de dicho vector será 2/5 veces el tamaño de la ventana gráfica en esa dirección.

El ***desplazamiento a medida*** tiene dos fases: interpretación de la solicitud del usuario y ejecución de esa solicitud. La primera fase va desde el instante en que el usuario pulsa el botón de desplazamiento del ratón y selecciona el punto fuente de desplazamiento, hasta el instante en que libera el botón de desplazamiento en el punto destino. La segunda fase procesa la información introducida y provoca el desplazamiento.

Cuando el usuario pulsa el botón de desplazamiento se provoca un evento que llama a una función de procesado. Esta función introduce un nodo especial en la lista de transformaciones, que se utiliza, como almacenamiento temporal de las coordenadas de la posición del puntero del ratón, en el momento en el que se pulsó el botón de desplazamiento. A esta posición la llamaremos punto original. Tras ésto, el usuario moverá el puntero, manteniendo el botón de desplazamiento pulsado. A medida que se va moviendo el puntero, se va borrando una línea entre la coordenada original y la coordenada anterior y se dibuja una línea entre la coordenada original y la actual. Esto se hace utilizando la función de dibujado de línea en modo O_exclusive. Cuando el usuario suelte el botón de desplazamiento, se toma la coordenada del puntero como coordenada final. El efecto que se ha de conseguir es que el punto final ocupe tras el desplazamiento la coordenada del punto original. Para ello, se restan las coordenadas del punto original a las del punto final, y la coordenada diferencia obtenida se introduce en la lista de transformaciones como nodo de desplazamiento. Tras ésto, se llama a la función de repintado para refrescar el esquema en pantalla.

b) Zoom.

El *zoom* se refiere a un escalado de la imagen. Desde el punto de vista del efecto que

producen existen dos tipos de zoom:

escalado "hacia adentro" (expresión inglesa "*zoom in*"), que provoca un aumento del esquema.

escalado "hacia afuera" (expresión inglesa "*zoom out*"), que provoca un empequeñecimiento del esquema.

Por otro lado, también existe otra división de los tipos de zoom atendiendo a la forma en que se realiza:

fijo: donde el factor de escalado que se introduce es constante al ser seleccionado un botón.

a medida: donde el usuario marca un área activa.

Aunque desde el punto de vista de proceso, no existen diferencias sustanciales entre el "*zoom in*" y el "*zoom out*" porque lo único que se hace es introducir un escalado en la lista de transformaciones, para el usuario estos dos conceptos son diferentes. Así mismo, si bien la segunda clasificación (*fijo, a medida*) no parece muy importante para el usuario, a nivel operativo es muy importante porque lleva un procesado muy diferente.

Cuando se realiza un escalado fijo, se introduce un factor en la lista de transformaciones constante. Se han utilizado los valores 2/3 para "*zoom out*" y 3/2 para "*zoom in*". Estos valores se han considerado los más útiles porque el escalado que producen es apreciable y no excesivo.

El proceso del escalado a medida tiene dos fases: **interpretación** de la solicitud del usuario y **ejecución** de dicha solicitud. La primera se refiere al tiempo que va desde que el usuario pulsa el botón de zoom del ratón (marcando el punto inicial del área activa) hasta el instante en que suelta dicho botón (marcando el punto final del área activa). En la segunda, calcula el escalado y desplazamientos implícitos en la solicitud e introduce ambas transformaciones en la lista de transformaciones del diseño. Téngase en cuenta, que este tipo de zoom lleva implícito un desplazamiento, puesto que el extremo inferior izquierdo del área activa debe coincidir con el extremo inferior izquierdo de la ventana gráfica tras el zoom.

Cuando el usuario pulsa el botón de zoom, el sistema lee en que coordenada gráfica está situado el puntero del ratón e introduce un nodo especial en la lista de transformaciones indicando este punto. Este nodo no contiene información de transformación gráfica ya que tan sólo es utilizado como almacenamiento temporal de la coordenada original. En este punto se dibuja una pequeña cruz para mejorar la realimentación visual del usuario. Cuando el puntero se mueve con el botón pulsado se borra el recuadro desde la coordenada anterior a la coordenada original y se dibuja un cuadro desde la coordenada original a la actual. De esta forma, va apareciendo una ventana que marca lo que va siendo el área activa en cada instante y sólo en el momento en el que se suelte el botón de zoom del ratón quedará fijada el área activa final. Para dibujar las líneas se utiliza el modo O_exclusive de tal suerte que siempre quedará la línea resaltada sobre lo que hay en la pantalla y además para borrarla tan sólo hay que volver a dibujar la misma línea con el mismo sistema.

Cuando se va a ejecutar la solicitud, se calculan los valores diferencias de los componentes

de la coordenada original y final del área activa. Si estos valores son muy pequeños, se desprecian y no se realiza escalado. Esto es debido, a que se considera que el usuario no es capaz de ajustar bien un factor de escalado muy grande a la primera y sería conveniente que lo realizara en varias aproximaciones sucesivas. Además, así evitamos zoom accidentales provocados por error de manipulación del ratón. También se desprecia los zoom en los cuales la coordenada final está fuera de la ventana gráfica. De esta forma introducimos un método para posibilitar que el usuario se pueda arrepentir de hacer el zoom una vez iniciada la operación.

En principio es imposible hacer coincidir el área activa seleccionada con la ventana gráfica final, porque, la relación ancho/largo del área activa y la ventana gráfica, serán diferentes en la gran mayoría de los casos y por supuesto, la relación ancho/largo del esquema no se debe tocar. Para el cálculo del factor de escalado se utilizan el siguiente algoritmo:

```
Desplazamiento_x=-Coordenada_original_x
Desplazamiento_y=-Coordenada_original_y
Insertar(Desplazamiento) en lista de transformaciones
Relacion_escalado=(Diferencia_x+Diferencia_y)/2
Relacion_ventana=(Ancho_ventana+Largo_ventana)/2
Si "zoom out" entonces
  Escalado_denominador=Relacion_escalado
  Escalado_numerador=Relacion_ventana
Sino
  Escalado_denominador=Relacion_ventana
  Escalado_numerador=Relacion_escalado
Fin si
Insertar(Escalado) en lista de transformaciones
```

Tras la inserción de estos nodos se recompone la lista de transformaciones para minimizar el número de nodos insertados y se refresca el gráfico representado en pantalla.

5.6.4.3.2 Selección de elementos dentro del diseño.

Este apartado trata del método utilizado para identificar una célula en el esquema de entre todas las posibles, pulsando el botón de selección de células en el ratón cuando el puntero del ratón está sobre ella.

El proceso para identificar la célula puede parecer complejo porque se ha de tener en cuenta que el dibujo no es estático. Esto es debido, a que el usuario pudo haber desplazado, o haber hecho zoom en diferentes ocasiones y con parámetros arbitrarios. Por ello, no debemos esperar que siempre en la misma coordenada esté el mismo elemento. En la práctica, el problema se descompone en dos partes: **identificar el punto sin transformaciones** e **identificar la célula que corresponde a dicho punto**.

Denominaremos **punto sin transformaciones** a *aquel punto que se refiera a un esquema virtual en el cual no se haya aplicado ninguna transformación gráfica*. En la mayoría de los casos, a los diseños se le habrán aplicado transformaciones gráficas, con lo que el primer problema se resolvería aplicando las transformaciones inversas al punto seleccionado, para convertirlo en un punto sin transformaciones. Para ello, ya se ha descrito la función InvTransform(). Con este punto sin transformaciones, el problema de la identificación de la célula se ha reducido considerablemente.

Todos los elementos que contengan descripción (SHEET, BLOCK, ARRAY y CELL) llevan en sus nodos un tamaño asociado. Podemos considerar que estamos dentro de cualquiera de ellos, si colocándolos en el punto de coordenadas (0,0), sin rotación y sin escalado, las

coordenadas del punto seleccionado están entre (0,0) y tamaño asociado al elemento. Esto nos da una pista de cómo ha de ser el método a seguir para identificar el elemento que estamos seleccionando.

En la práctica, podemos considerar que seleccionamos dentro de un SHEET, no cuando estemos dentro de las coordenadas posibles para ese SHEET, sino cuando seleccionamos dentro de la ventana gráfica asociada a el SHEET. El sistema, por otro lado, tratará de profundizar al máximo en la identificación del elemento. Esto significa, que penetrará en todos los niveles jerárquicos necesarios, hasta identificar una célula que se toma como elemento básico. No se considera una selección como correcta si no se identifica a ninguna célula. Aún así, el programa de búsqueda devolverá el nombre asociado al camino jerárquico que se está seleccionando.

Para comprender el método utilizado para la identificación de la célula, se ha de comprender cómo era el método para dibujar el esquema. Cuando se pretendía dibujar un esquema, lo que se hacía en definitiva, era ir recorriendo la estructura de descripción e ir insertando nodos de transformación gráfica por cada comando que tuviera significado gráfico, y, cuando se encontraran instrucciones efectivas de dibujo (tales como dibujo de líneas, rectángulos, ...) entonces se transformaban sus coordenadas según la lista de transformaciones existentes en el momento de procesado y se generaban las llamadas a las primitivas gráficas del *OpenWindows* con las coordenadas ya transformadas. Esto nos da otra pista acerca del método que se puede utilizar para identificar elementos.

Lo anteriormente expuesto nos dice, que si recorremos la estructura de descripción generando los nodos de transformación gráfica de igual forma que se haría cuando se va a dibujar la descripción, si al invocar el procesado de elementos con descripción realizáramos la transformación inversa del punto sin transformaciones con la lista existente en ese momento, este punto procesado debería quedar dentro de los márgenes del elemento si originalmente fue seleccionado dentro de ese elemento. Es decir, que tendremos que escribir funciones gemelas a las de dibujo pero orientadas al reconocimiento de células. El nombre del elemento que se genera está relacionado con la combinación de llamadas de transformación realizadas a través de los diferentes niveles jerárquicos.

Se han implementado las siguientes funciones que en conjunto realizan tal proceso:

RasterDesign(BloDes *d, char respuesta[], char pathname[], int x,y);

Esta es la función global para identificación de elementos. Desencadena una sucesión de llamadas, similar a la que generaba la función **GrDesign()** atendiendo a la descripción que contiene, tratando de identificar qué elemento está ubicado en la coordenada (*x,y*) que se le pasa. Obsérvese que se da por sabido el SHEET que genera esa llamada (parámetro *d*). Esto es cierto, puesto que como se comentó antes, éste habrá sido identificado por la ventana gráfica en la que se produjo la selección. Los parámetros *respuesta[]* y *pathname[]* son parámetros de salida. El primero nos identifica la célula con el nombre completo de variable asociada a ella, incluyendo índices de *ARRAY*s que tuviera. El parámetro *pathname[]* nos da un nombre de variable sin *ARRAY*. Su utilidad es poder identificar a través de él la descripción del modelo de la célula que se selecciona. Esta función devuelve el valor **OK** en caso de éxito y el valor **FAIL** en caso de fallo.

RasterBlock(BloDes *d,*b, char respuesta[], char pathname[], int x, y, CmList *l);

Esta función es igual a la anterior excepto, por el hecho de que se le pasa un parámetro adicional *b*, que indica cual es el bloque que se desea rastrear. Otra diferencia que se ha de tener en cuenta es que esta función no modifica el nombre de la variable en proceso, puesto que el nombre de un bloque nunca aparece como parte del nombre de una variable. A través del parámetro *l* se le pasa la lista de transformaciones gráficas asociada a este *BLOCK* en esta llamada.

RasterPlace(BloDes *d, Place *p, char respuesta[], char pathname[], int x, y, CmList *l);

Esta función procesa un *PLACE* (parámetro *p*) en cualquier descripción. Genera los nodos de desplazamiento, rotación y los de escalado selectivo si fuesen pertinentes. Además, realiza una llamada al tipo de función de búsqueda adecuado al tipo de elemento que lleve asociado. Si la búsqueda tuviese éxito, añadiría su nombre a los nombres de variables devueltos. A través del parámetro *l* se le pasa la lista de transformaciones gráficas.

RasterArray(BloDes *d, Array *a, char respuesta[], char pathname[], int x, y, CmList *l);

Esta función procesa un *ARRAY* (parámetro *a*) en cualquier descripción. Genera un nodo de desplazamiento que inserta en la lista de transformaciones. Luego de ésto, realiza dos bucles anidados en los cuales se generan llamadas al procesado del elemento que tiene asociado, con los índices que recorren el *ARRAY*. Antes de cada llamada se actualiza el nodo desplazamiento que se insertó para ajustarlo al valor que le corresponde en ese momento. Si alguna llamada tuviera éxito, entonces se acabaría el bucle y esta función insertaría los índices correspondientes a esa llamada, al nombre de variable. A través del parámetro *l* se le pasa la lista de transformaciones gráficas asociada.

RasterCell(BloDes *d, Cell *c, char respuesta[], char pathname[], int x, y, CmList *l);

Esta función procesa un *CELL* (parámetro *c*) en cualquier descripción. Cuando es llamada realiza la transformación inversa de las coordenadas (*x,y*) con la lista de trasformaciones (parámetro *l*). Luego de este paso, se ve si esta coordenada transformada queda dentro del elemento *CELL*, es decir, que los valores de la coordenada asociada deben quedar dentro del valor (0,0) y del tamaño de la célula. Esta última condición se considerará la condición de éxito.

Cuando se realiza una selección con éxito, a la vuelta se toma el *pathname[]* para buscar la descripción de la célula ubicada bajo el puntero. Con esta descripción se abre una subventana que contiene una lista de variables de la célula. Por otro lado, con la *respuesta[]* podemos localizar la dirección física de memoria en la cual ese elemento está guardando sus variables. De esta forma basándonos en los offsets que nos indican cada una de las variables de la célula podemos identificar exactamente la dirección de cada variable en memoria y así

podemos leerlas para presentar en la subventana el valor actual que contiene y si el usuario cambiara uno de estos valores también podríamos modificar la variable.

5.6.4.3.3 Otras funciones de control del entorno.

Estas funciones de control son: **conmutación de modo "*zoom in*" a "*zoom out*", control de visibilidad** y **vista completa.**

La conmutación de modo "*zoom in*" a "*zoom out*" consiste en cambiar un bit asociado al diseño que nos indica si el modo de zoom a medida es "*zoom in*" o "*zoom out*". Para tal caso existe un botón en el panel de la ventana del diseño, que en cada momento nos indica gráficamente, cual de los dos modos está activo.

El control de visibilidad nos abre una subventana que contiene una lista de niveles, su situación dentro del diseño (activado, desactivado) y una muestra gráfica del color y del patrón de relleno asociado a ese nivel. El usuario puede cambiar el estado de los niveles (excepto el del 0, que se toma como fondo). Al cerrarse la ventana se llama a la función de repintado la cual respeta los niveles activos a la hora de dibujar.

La vista completa borra la lista de transformaciones gráficas asociada al diseño y calcula una lista de transformaciones gráficas nueva, con la cual, sea cual sea el tamaño en pixels del diseño, éste será totalmente visible utilizando de forma óptima el tamaño de la ventana gráfica. El proceso para realizar esta función sigue el método de la *coordenada débil* que consiste en detectar cual de los dos tamaños (horizontal o vertical) es mayor respecto al tamaño (horizontal o vertical) de la ventana. La división de esa *coordenada débil* por el tamaño de la ventana, nos dará el factor de escalado a utilizar.

Conclusiones y líneas futuras.

Conclusiones.

Como resultados de los trabajos realizados en la presente tesis, podemos señalar que se ha desarrollado un nuevo sistema de ayuda a la concepción, diseño y simulación de arquitecturas con alto grado de paralelismo, que además cubre perfectamente objetivos de formación docente. Resaltando que una gran parte de las aportaciones son aplicables a cualquier otro tipo de arquitecturas. La evaluación del entorno mediante el trabajo con distinto tipo de arquitecturas de referencia permite asegurar la facilidad y flexibilidad de uso. Además la característica de ser un entorno abierto permite la rápida incorporación de nuevas prestaciones.

A continuación presentamos un resumen de las contribuciones consideradas de mayor interés.

* De ha desarrollado el Lenguaje de Descripción de Arquitecturas Paralelas (LDAP). Los elementos a manejar son células, arrays, bloques y sheets. Posee la capacidad de dar información sobre las líneas físicas de entrada/salidas de las células, lo que va permitir que a la hora de la declaración de estructuras tipo array, puedan realizarse las autoconexiones entre células adyacentes. Dispone de comandos pensados para la declaración de conexiones entre estructuras compleja. Además nos ayuda a la simulación de estas arquitecturas con simuladores estándar, ya que a partir de una misma descripción podemos generar ficheros de entrada a distintos simuladores.

* Se ha desarrollado una librería de funciones C que realiza todas las funciones del lenguaje LDAP. Esta metodología de trabajo permite al usuario la creación de sus propios programas o generadores de arquitecturas.

* Se ha desarrollado el Lenguaje de Descripción Visual (LDV). Es un lenguaje que permite la descripción gráfica de las arquitecturas diseñadas con LDAP.

* Se ha desarrollado el Lenguaje de Intercambio de Variables (LIV). Su función es la permitir la comunicación entre la herramienta de visualización y monitorización (SiMon) y los simuladores que se vayan a utilizar.

* Se ha desarrollado la herramienta **GeneSis**, que incorpora el analizador léxico del lenguaje LDAP, además de otras utilidades que facilitan el trabajo de definición de arquitecturas. A partir de la descripción LDAP de entrada se ocupa de la generación de los ficheros de salida.

* Se ha desarrollado la herramienta **SiMon**, encargada de la monitorización y control de las arquitecturas descritas, mediante un interfase gráfico a base de ventanas independientes. Incorpora los interpretes de los lenguaje LDV y LIV. Y se encarga además del intercambio de variables con el simulador que se utilice.

* La estructura de las distintas herramientas está desarrollada de forma modular, lo que posibilita la incorporación de nuevos elementos y la conexión a diferentes simuladores comerciales.

Esta tesis pretende agilizar los primeros estadios de concepción del diseño de arquitecturas con alto grado de paralelismo, facilitando el camino a la simulación y realización física del sistema mediante herramientas estándar. El trabajo con EASAP significa un nuevo enfoque de trabajo, y es una ayuda al diseñador en los puntos siguientes:

Ayuda a la concepción:

Podemos de una forma rápida realizar una descripción estructural de una arquitectura paralela y visualizar ésta gráficamente de forma muy rápida, lo que nos facilita la verificación y corrección de errores.

El tener de forma rápida una imagen de la arquitectura descrita acelera las etapas iniciales del diseño.

En este tipo de arquitecturas, la capacidad de ver evolucionar los flujos de datos sobre una representación gráfica de la arquitectura descrita, da al diseñador más y mejor información que la suministrada por un diagrama lógico o tablas de estados.

La característica de ser un entorno abierto nos permite la conexión de programas que implementen algoritmos generadores de arquitecturas a partir de descripciones algorítmicas. Esto nos permitirá concentrarnos en el algoritmo utilizado.

La posibilidad de mostrar simultáneamente arquitecturas y flujos de datos nos da una herramienta de formación sumamente potente.

Ayuda al diseño:

Una vez realizada la descripción estructural de una arquitectura objeto de estudio en nuestro entorno, se puede traspasar directamente al entorno CAD en el cual se continuará con las labores de realización física, evitando tareas de muy bajo nivel o muy monótonas.

Podemos incorporar nuevas funciones de evaluación tecnológica de las arquitecturas VLSI descritas, con lo que al visualizar éstas, podemos tener además una estimación del área, potencia, etc., lo que supone tener información valiosa sobre variables a tener en cuenta a la hora de optar por arquitecturas alternativas.

Ayuda a la simulación:

Al incorporar la descripción estructural de nuestra arquitectura a la base de datos del entorno CAD con el que trabajemos, podemos realizar su simulación con el simulador que se encuentre soportado.

La herramienta de visualización gráfica propuesta, puede ser conectada a distintos entornos facilitando las tareas de simulación y comprensión de los resultados.

Líneas abiertas.

En el Centro de Microelectrónica Aplicada de la Universidad de Las Palmas se está trabajando actualmente en la ampliación de los conceptos presentados en esta tesis, dentro de las siguientes líneas:

Se está trabajando en la optimización del intérprete del lenguaje LDAP.

Se está ampliando el lenguaje LDV para permitir la declaración conjunta de los elementos gráficos actuales con nuevos elementos como gráficos Postscript, gráficos HPGL o bitmaps.

Se están ampliando las primitivas gráficas del visualizador SiMon para permitir visualizar el cambio de variables mediante nuevos elementos gráficos como barras indicadoras, indicadores digitales y analógicos, etc..

Se está creando una librería de elementos gráficos en formato LDV específica para la monitorización y control de procesos industriales remotos como parte de proyectos concertados con empresas.

Se está ampliando el lenguaje LIV para un control total de SiMon desde un proceso remoto.

Está en preparación diversos programas con carácter docente, que mediante la utilización del LDAP compilado generen y visualicen distintas arquitecturas clásicas VLSI con el objetivo de reforzar la comprensión del funcionamiento de estas arquitecturas por parte del alumno.

Está en preparación la integración de las herramienta desarrolladas dentro del entorno Mentor Graphics.

Las líneas que se consideran de gran interés, pero que en el momento de la publicación de esta tesis no han sido abordadas son las siguientes:

Está en estudio la realización de un módulo de evaluación tecnológica de arquitecturas digitales CMOS.

Está en estudio la realización de un módulo de evaluación tecnológica de arquitecturas digitales GaAs.

Se podrán crear nuevos módulos de salidas para otros simuladores, sin más que añadir las funciones de librería apropiadas para ello.

Referencias.

[Anna86] Annaratone, M., et al., "WARP Architecture and Implementation", Proceedings 13th International Symposium on Computer Architecture, June 1986, pp. 346-356.

[Arms83] Armstrong, J.R., "Chip Level Modeling and Simulation", Simulation, October 1983, pp. 141-148.

[Arms88] Armstrong, J.R., "Chip Level Modeling with HDLs", IEEE Design and Test of Computers, February 1988, pp. 8-18.

[BaWo83] Baugh. C.R., and Wooley, B.A. "A two's complement parallel array multiplication algorithm", IEEE Trans., 1983, C-22, pp. 1045-1247.

[BeNe71] C.G. Bell and Newell, "Computer Structures: Readings and Examples", New York, NY: McGraw-Hill, 1971.

[BoPi92] Dominique Borrione, Robert Piloty, et al., "Three Decades of HDLs - Part2: Conlan Through Verilog", IEEE Design and Test Computer, September 1992, pp. 54-63.

[BrHaS90] R.K. Brayton, G.D. Hachtel, and A.L. Sangiovanni-Vincentelli, "Multilevel Logic Syntesis", Proc. of the IEEE, 1990, Vol. 78, No. 2, pp. 264-300.

[CaWh82] McCanny, J.V., and McWhirter, J.G.:"Implementation of signal processing function using 1-bit systolic arrays", Electron, Lett., 1982, 18, (6), pp. 241-243.

[CaWh83] McCanny, J.V., and McWhirter, J.G.:"Bit level systolic array circuit ", Electron, Lett., 1982, 18, (6), pp. 241-243.

[CaWh84] McCanny, J.V., McWhirter, J.G., and Wood, K.W., "Optimized bit level systolic array for convolution", IEE Proc. F, Commun., Radar & Signal Process., 1984, 131, (6), pp. 632-637.

[CaWo83] McCanny, J.V., Wood, K.W., McWhirter, J.G. and Oliver, C.J.: "The relationship between word and bit level systolic arrays as applied to matriz x matriz multiplication", SPIE Real Time Signal Process. V.I, 1983, 431, pp. 114-120.

[Chu92] Yaohan Chu, Donald L. Dietmeyer, James R. Duely, et al., "Three Decades of HDLs - Part1: CDL Through TI-HDL", IEEE Design and Test Computer, June 1992, pp. 69-81.

[CoPa83] Corry, A.G., and Patel, K., "Architecture of a CMOS correlator", Proc. IEEE Symp. on Circuits and Systems, 1983, pp. 522-525; GEC J. Res., 1983, 1, pp 35-38.

[Fly72] Flynn, M.J., "Some computer organizations and effectiveness", IEEE Trans. Comput., 1972, C-34. pp. 948-960.

[FoKu80] Foster, M.J., and Kung, H.T., "The design of special purpose VLSI chips", IEEE Computer, 1980, 13, pp. 26-40.

[GaKu83] D.D. Gajski and R.H. Kuhn, "New VLSI Tools", Computer, Dec. 1983, Vol. 16, No. 12, pp. 11-14.

[Geus92] Aart J. de Geus, "VHDL: Toward a Unified View of Design", IEEE Design and Test of Computers", June 1992, pp. 8-17.

[HaSp90] D. S. Harrison, A. R. Newton, R. L. Spickelmier and T. J. Barnes, "Electronic CAD Frameworks", Proc. of IEEE, 1990, Vol. 78, No. 2, pp. 393-417.

[HuKuh85] T.C. Hu and E.S. Kuh, Eds., "VLSI Circuit Layout: Theory and Design", New York:IEEE Press, 1985.

[HwBr88] Hwang, K., Briggs, F.A., "Arquitectura de computadores y procesamiento paralelo", McGraw-Hill, 1988.

[HwCh80] Hwang, K., and Cheng, Y.H.: "VLSI computing structures for solving large-scale linear system of equations", Proc. Int. Conf. on Parallel Processing, Aug. 1980, pp, 217-227.

[HwCh82] Hwang, K., and Cheng, Y.H.: "Partitioned matrix algorithms for VLSI arithmetic systems", IEEE Trans., 1982, C-31, pp 1215-1224.

[KnPa85] D.W. Knapp and A.C. Parker, "A Unified Representation for Design Information", in 7th Int'l Conf. on CHDL. Amsterdam, The Netherlands: North-Holland, 1985, pp. 337-353.

[Kug88] Kugelmass, S.D., Steiglitz, K. "A Probabilistic Model for Clock Skew", Proceedings International Conference on Systolic Arrays, San Diego, May 1988, pp.545-554.

[KuhOh90] Ernest S. Kuh and Tatsuo Ohtsuki, "Recent Advances in VLSI Layout", Proc. of the IEEE, 1990, Vol. 78, No. 2, pp. 237-263.

[Lem89] Karin Lemmert, "-SYS^3- A Systolic Synthesis System Around KARL", Ph.D Thesis, Kaiserslautern 1989.

[Maly90] Wojciech Maly, "Computer-Aided Design for VLSI Circuit Manufacturability", Proc. of IEEE, 1990, Vol. 78, No. 2, pp. 356-392.

[McPaC90] Michael C. McFarland, Alice C. Parker and Raul Camposano, "The High-Level Synthesis of Digital Systems", Proc. of IEEE, 1990, Vol. 78, No. 2, pp. 301-318.

[MeCo80] C. Mead and L. Conway, "Introduction to VLSI Systems", Addison-Wesley, 1980.

[Men92] "Mentor Graphics System Overview Manual", Sotware Versión 8.1, Mentor Graphics Corporation 1992.

[MiLB88] Michael L. Bushnell, "Design Automation, Automated Full-Custom VLSI Layout Using the ULYSSES Design Environment", Academic Press, INC., 1988.

[MiSaA87] G. De Micheli, A. Sangiovanni-Vincentelli and P. Antognetti, Eds., "Design Systems for VLSI Circuits", Dordrecht, The Netherlands: Martinus Nijhoff Publishers, 1987.

[Oht86] T. Ohtsuki, Ed., "Layout Design and Verification", Amsterdam: North Holland, 1986.

[PoKe80] Powell, N.R., and Kerwin, J.M.: "Signal processing with bit serial word parallel architectures", SPIE Real Time Signal Process. I, 1978, 54, pp 98-104.

[SeLee87] C. Sechen and K-W. Lee, "An improved simulated anneling algorithm for row-based placement", in Digest of Tech. Papers, ICCAD, 1987, pp. 478-481.

[SeSa86] C. Sechen and A. Sangiovanni-Vincentelli, "Timber Wolf 3.2: A new standard cell placement and global routing package", in Proc. 23rd DAC, 1986, pp. 432-439.

[StMi89] Steven S. Leung and Michael A. Shanblatt, "ASIC System Design with VHDL: A Paradign", Kluwer Academic Publishers, 1989.

[StLH92] Stanley L. Hurst, "Custom VLSI Microelectronics", Prentice Hall, 1992.

[Stro89] A. J. Strojwas, "Design for Manufacturability and Yield", Proc. 26th Automation Conf.,1989, pp. 454-459.

[ThMo91] D.E. Thomas and P.R. Moorby, "The Verilog Hardware Decription Language", Kluwer Academic Publisher, Boston 1991.

[TonH87] Tony Holden, "Knowledge based CAD and Micro-Electronics", North-Holland, 1987.

[UrWo84] R.B. Urquhard & D. Wood, "Systolic matrix and vector multiplication methods for signal processing", IEE Proc. Vol. 131. Pt. F. No. 6, October 1984.

[Ur83] Urquhart, R,B, "VLSI architectures for the linear discriminant classifier", Proc. IEEE Computer Society Workshop on Computer Architectures for Pattern Analysis and Image Database Management (CAPAIDM), October 1983, pp. 227-232.

[Ver90] Verschueren, A.C.,"An Object Oriented Design and Simulation System for VLSI", Microprocessing and Microprogramming, Volume 30, Numbers 1-5, August 1990, PP. 241-246.

[Ver92] Verschueren, A.C.,"An Object-Oriented Modelling Technique for Analysis and Design of Complex (Real-Time) Systems", Ph.D thesis, Eindhoven 1992.

[WaTho85] R.A. Walker and D.E. Thomas, "A Model of Design Representation and Synthesis", in Proc. of the 22nd Design Automation Conf., New York, NY: ACM/IEEE, 1985, pp. 453-459.

[WhCa82] McWhirter, J.G. McCanny. J.V. and Wood, K.W. "Novel multibit convolver/correlator chip based on systolic array principles", SPIE, Real Time Signal Processing V, 1982, 341, pp. 66-73.

[WoEv83] Wood. D., Evans, R.A., and Wood, K.W., "An 8 bit serial convolver chip based on a bit level systolic array". Proc. of the Custom Integrated Circuit Conference, May 1983, pp 256-261.