Universidad de Granada

Generación automática de sistemas bioinspirados de visión en hardware reconfigurable

Antonio Martínez Álvarez

MEMORIA DE TESIS DOCTORAL

Directores: Francisco José Pelayo Valle
Leonardo Maria Reyneri

Mayo de 2006

Dpto. Arquitectura y Tecnología de Computadores

Editor: Editorial de la Universidad de Granada
Autor: Antonio Martínez Álvarez
D.L.: Gr. 753 - 2006
ISBN: 84-338-3812-1

D. Francisco José Pelayo Valle, Catedrático de la Universidad de Granada, y D. Leonardo Maria Reyneri, Catedrático del Politécnico de Turín,

CERTIFICAN

Que la memoria titulada **GENERACIÓN AUTOMÁTICA DE SISTEMAS BIO-INSPIRADOS DE VISIÓN EN HARDWARE RECONFIGURABLE**, ha sido realizada por *D. Antonio Martínez Álvarez* bajo nuestra dirección en el Politécnico de Turín y en el Departamento de Arquitectura y Tecnología de Computadores de la Universidad de Granada, para optar al grado de Doctor y a la mención especial de Doctor Europeo.

Granada a 22 de Marzo de 2006

Fdo. Francisco José Pelayo Valle	Fdo. Leonardo Maria Reyneri

A Esther y mis padres.

Uno es para siempre responsable
de lo que domestica.
Antoine de Saint-Exupéry

Agradecimientos

Agradezco la colaboración directa o indirecta a todas las personas que de algún modo han contribuido en la elaboración de la presente tesis doctoral.

Mi más enérgico y sincero agradecimiento a mis directores de tesis Francisco José Pelayo Valle y Leonardo Maria Reyneri. A Francisco por sus acertados comentarios sobre todo mi trabajo, por enseñarme mediante sus correcciones y ejemplos cómo se debe publicar un artículo científico y por su dedicación desmedida al trabajo de investigación que hemos compartido. A Leonardo por la enorme hospitalidad que mostró conmigo durante el trabajo doctoral en el *Politecnico di Torino*, en el que experimenté su tremenda dedicación e intensidad en el trabajo conjunto, que sin duda tanto me ha beneficiado. Extiendo también mi gratitud a la familia de Leonardo, especialmente a Daniella, su mujer, por su enorme hospitalidad y cariño. A todos nuevamente, *grazie mile!*.

Un agradecimiento especial al resto del equipo humano del proyecto *CORTIVIS* en Granada (Christian y Samuel), con los que tantas cosas he compartido y sin el que hubiera sido imposible llevar nuestra investigación a buen puerto. Un recuerdo especial a Alberto Prieto, Julio Ortega, Richard y todos mis *siempre* compañeros del Departamento de Arquitectura y Tecnología de Computadores de la Universidad de Granada.

Agradezco también la financiación de este trabajo por parte de los proyectos europeos de investigación *CORTIVIS* y *SPIKEFORCE*, al proyecto *DPI2004-07032* del Plan nacional de I+D+I y por último el Plan Propio y el Vicerrectorado de Investigación y Tercer Ciclo de la Universidad de Granada.

A mis nuevos compañeros del Departamente de Tecnología Informática y Computación de la Universidad de Alicante, por su apoyo, ánimo y compañía en los últimos meses de redacción de este documento.

A Paco Alonso y Juan Eloy por su gran ayuda en la redacción en LaTeXde este documento.

Y por último, mi gratitud más cariñosa y agradecida a mi novia Esther y por supuesto a mis padres, a los que tantos malos ratos he hecho pasar.

Grazie mile di cuore!

Abstract

Abstract

This dissertation focuses on the design of bioinspired hardware/software visual processing systems. The presented work has been motivated by the European Project *CORTIVIS* (QLK6-CT-2001-00279). The main purpose of *CORTIVIS* is the design of a cortical visual neurophosthesis able to automatically deliver neuro–stimulation currents to the visual area of the brain cortex in blind individuals.

The presented solution performs a top–down design flow of a retina model, taking into account different levels of abstraction.

The result is a general system for specification, simulation, testing and synthesis of general-purpose computer vision models with a special emphasis on a custom optimization strategy. The main tasks carried out by this work are set out as:

- Definition of a retina model.
- Design of a high–level simulator of bioinspired vision models with the possibility of contrasting biological and synthetic results.
- Development of a system able to perform the automatic generation of bioinspired retina–like processing hardware, from a high level specification of the model defined using the previous simulator. The high level specification is mapped on reconfigurable hardware considering a number of custom optimizations.

The proposed simulation and synthesis platform generates optimal circuit descriptions, using the *IEEE* standard language *VHDL*, which have been tested on various *FPGA* technologies and with different lower level synthesis tools.

Resumen

La presente tesis doctoral aborda el diseño hardware/software de sistemas de procesamiento de la visión. El trabajo ha sido motivado por el Proyecto

Europeo *CORTIVIS* (QLK6-CT-2001-00279), cuyo principal objetivo es el diseño de una neuroprótesis cortical capaz de producir de forma automática, corrientes de neuroestimulación en la corteza visual de personas invidentes.

La solución que se aporta lleva a cabo un diseño descendente (*top–down*) de un modelo de retina, teniendo en cuenta distintos niveles de abstracción.

El resultado es un sistema diseñado para la especificación, simulación, comprobación y síntesis de modelos computacionales de visión que hace un énfasis espacial en una particular estrategia de optimización. Las principales tareas llevadas a cabo en el trabajo doctoral se enmarcan en los siguientes ítems:

- Definición de un modelo de retina.
- Diseño de un simulador funcional de modelos de visión bioinspirados con la posibilidad de contrastar los resultados biológicos y los sintéticos.
- Desarrollo de un sistema capaz de llevar a cabo la generación automática de sistemas bioinspirados de visión (como una retina) en hardware, a partir de especificaciones en alto nivel del sistema dadas por el anterior simulador. Esta especificación del modelo se transporta a hardware reconfigurable considerando ciertos procesos de optimización.

La plataforma de simulación y síntesis genera descripciones de circuito digital óptimas, usando el lenguaje estándard de *IEEE VHDL*, que han sido evaluadas de forma satisfactoria en varias tecnologías *FPGA* y probadas con diferentes herramientas de síntesis lógica.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En este primer capítulo introductorio, se resume a modo de prólogo el planteamiento del trabajo de tesis y el contenido de la presente memoria. La sección 1.1 introduce los objetivos que han fundamentado el desarrollo del trabajo doctoral y describe los propósitos generales de la tesis doctoral. Posteriormente, la sección 1.2 presenta el marco de desarrollo del trabajo doctoral y define el proyecto europeo de investigación en el que se ha fundamentado la misma. La sección 1.3 resume la metodología de trabajo seguida. En la sección 1.4 se define la articulación por capítulos de la presente memoria y por último, la sección 1.5 señala las convenciones tipográficas seguidas en la redacción de la misma.

1.1. Introducción

La presente tesis doctoral aborda la problematica del diseño de sistemas hardware/software bioinspirados de visión. La solución propuesta, fundamentada en los resultados del proyecto europeo *CORTIVIS* (*CORTIcal Visual Neuroprosthesis for the Blind*), ha integrado la definición y simulación de alto nivel del prototipo de visión, con la obtención mediantes técnicas de síntesis automática hardware/software del modelo de visión final en hardware reconfigurable.
A continuación se presenta la motivación del trabajo doctoral, fundamentando la articulación de cada una de las tareas que han tenido lugar.

1.2. Motivación y planteamiento del problema

El avance en los últimos años en áreas de conocimiento tales como la ingeniería electrónica, la ingeniería del software, las neurociencias, ha dado lugar a nuevas disciplinas, la *ingeniería neuromórfica* y la *neuroingeniería*, que mediante la convergencia de los esfuerzos de las anteriores ciencias, emergen con un marcado carácter interdisciplinar, para estudiar y resolver problemas concretos mediante sistemas artificiales o conectando con el nervio biológico. La *Inge-*

niería Neuromórfica (consultar la sección 3.4 del capítulo 3) plantea el diseño de sistemas artificiales de computación que heredan ciertas estructuras y propiedades de un sistema nervioso biológico[1]. Por otra parte, la neuroingeniería aborda el estudio y desarrollo de *interfaces* con el sistema nervioso biológico con el fin de proporcionar una interacción útil con el mismo.

El proyecto europeo *CORTIVIS*, descrito de forma más detallada en la sección 3.5, está fundamentado por los principios de la ingeniería neuromórfica y la neuroingeniería define su principal propósito de acuerdo al siguiente item:

> *Llevar mediante una o varias cámaras de vídeo, una descripción visual del entorno a la corteza visual humana, con objeto de restituir o mejorar parcialmente la visión a personas invidentes. Para este cometido se diseñará una neuroprótesis de corteza visual humana que utilice una matriz de microelectrodos para interaccionar con los centros de visión del cerebro.*

En el marco de *CORTIVIS*, se ha llevado a cabo una investigación en la interfaz de varias ciencias como las ingenierías electrónica e informática, las matemáticas la neurofisiología y neurocirujía, la histología etc. Todas estas disciplinas han aportado algo valioso y fundamental en el trabajo de especificación, modelado, simulación y en definitiva el diseño definitivo de la neuroprótesis; empeños todos ellos en los que el doctorando ha tomado parte activa como miembro del proyecto de investigación.
El objetivo concreto de diseñar un chip capaz de emular el comportamiento de la retina humana y otras capas del tracto óptico ha llevado a proponer una solución portable y de bajo coste que fuera a su vez altamente configurable, dadas las características particulares de cada paciente. La cualidad de la neuroprótesis de ser altamente configurable y parametrizable descansa en el hecho de que la corteza visual biológica, así como el resto del tracto óptico biológico, presenta innumerables peculiaridades propias de cada indivíduo que hay que modelar convenientemente para una correcta adecuación neurológica.

Los sistemas basados en software (computadoras, *DSPs*, microcontroladores, ...) suelen ofrecer una buena relación coste/rendimiento. No obstante, existe una gran variedad de aplicaciones en las que, como es el caso de la neuroprótesis que nos ocupa, los requisitos de velocidad, rendimiento, consumo etc. hacen tender la balanza a una realización completa en *hardware* o híbrida (*Software/Hardware*).

Por este motivo, la tecnología empleada en los primeros prototipos de retina ha sido la tecnología *FPGA* (descrita con detalle en la sección 2.2.1) que flexibiliza notoriamente el *transporte* de algoritmos a un soporte hardware.
En resumen, un objetivo del presente trabajo de tesis doctoral consiste en integrar distintos conocimientos propios de la neuroingeniería y la ingeniería neuromórfica, para modelar correctamente una retina cuyo comportamiento deberá ser sintetizado en hardware de forma conveniente.

La traslación de un modelo funcional de retina, y en general de cualquier sistema de procesamiento de información visual, a una implementación en circuito electrónico digital, requiere de un esfuerzo de diseño y validación muy

considerable, y de una gran especialización en diseño de hardware. Para soslayar estas dificultades, el segundo y principal objetivo de la tesis ha sido el diseño e implementación de una plataforma de síntesis automática de modelos de visión en hardware digital reconfigurable.

1.3. Metodología de trabajo

La metodología del trabajo ha venido marcada por el desarrollo del proyecto *CORTIVIS*. En los 3 años y medio de duración del proyecto se han llevado a cabo las siguientes tareas relacionadas directamente con la presente tesis doctoral:

- Definición del modelo de retina.
- Desarrollo de un simulador de sistemas de visión bioinspirada, con la posibilidad de contrastar los resultados biológicos con los generados de forma sintética.
- Desarrollo de un prototipo de retina en hardware reconfigurable.
- Desarrollo de un sistema de síntesis automático de sistemas digitales de visión, para transladar el modelo validado de retina a una descripción en hardware reconfigurable.

1.4. Estructura de la tesis

La memoria de tesis doctoral se ha articulado en 10 capítulos, 4 apéndices y una sección bibliográfica. En esta estructura se ha intentado ofrecer capítulos bien conexos que posibiliten tanto una lectura lineal de la memoria, como una lectura selectiva.
Con la voluntad de favorecer el análisis y la búsqueda de conceptos clave, se ha adjuntado al final de la memoria un índice alfabético de los términos más representativos de que se trata, obteniéndose en la mayoría de los casos una definición del concepto en cuestión.
La descripción del núcleo principal del trabajo de la tesis se ha organizado en 4 capítulos distintos. Un capítulo para introducir dicho núcleo (el capítulo 4) y los tres siguientes (capítulos 5, 6 y 7) para desarrollar cada uno de los diferentes aspectos del trabajo (simulación, modelado hardware y síntesis automática).

1.4.1. Organización

A continuación se presenta la organización por capítulos de la tesis doctoral:

- Capítulo 1. *Introducción*: Breve introducción y organización de la presente memoria de tesis doctoral, incluyendo las convenciones tipográficas.

- Capítulo 2. *Problemática del diseño automático de sistemas digitales orientados a visión*: En este capítulo se reseña el *estado del arte* del diseño automático de sistemas digitales y se hace una comparativa de las diferentes herramientas relacionadas con esta disciplina, concretamente se describe con más detalle el entorno de diseño *CodeSimulink*, hacia el que se ha dirigido una parte de los esfuerzos del presente trabajo.

- Capítulo 3. *Situación actual. Modelo de retina*: Este capítulo presenta la retina biológica desde el punto de vista histológico y fisiológico y repasa los principales modelos de la misma, así como el modelo que finalmente se ha escogido para mimetizar el comportamiento retiniano. También se explora el espectro de prótesis neurológicas haciendo especial énfasis en los neuroimplantes visuales.

- Capítulo 4. *Plataforma de diseño de sistemas de visión*: El cuarto capítulo introduce la plataforma de diseño de sistemas de visión y articula su planteamiento en los siguientes tres capítulos.

- Capítulo 5. *Simulación funcional de modelos de visión*: Este capítulo presenta los métodos de simulación empleados para diseñar sistemas de visión e introduce *Retiner*, una herramienta de diseño y validación de sistemas bioinspirados de visión.

- Capítulo 6. *Implementación hardware*: Descripción de la implementación hardware del sistema de visión, presentando cada uno de los módulos.

- Capítulo 7. *Generación automática del sistema hardware/software. La herramienta HSM.*: Se presenta en este capítulo la estrategia de generación automática hardware/software del sistema de visión. Describiendo cómo se ha diseñado el proceso de síntesis y las distintas optimizaciones.

- Capítulo 8. *Resultados y validación experimental*: Este capítulo expone una serie de resultados obtenidos mediante la plataforma de diseño explorando cada uno de sus componentes, mostrando pruebas de simulación y síntesis de un sistema de visión bioinspirado completo y de cada una de sus partes.

- Capítulo 9. *Conclusiones y principales aportaciones*: El último capítulo de la memoria aborda las conclusiones de la misma, resumiendo el trabajo de tesis doctoral y presentando las líneas de investigación futuras.

- Capítulo 10. *Conclusions and main contributions*: Este capítulo es una traducción al inglés del anterior con objeto de, según las normas vigentes de la comisión de doctorado, poder optar a la mención especial de doctorado europeo.

- Apéndice A. *Reseña matemática*: En este apéndice se puede consultar el listado de las definiciones de funciones matemáticas que se han empleado a lo largo de la memoria.
- Apéndice B. *Código fuente del módulo convolver*: El primer apéndice muestra el código *VHDL* del módulo hardware más importante de toda la plataforma de diseño, comentado debidamente para distinguir claramente los tres esquemas de computación de que consta (paralelo–paralelo, paralelo–serie, serie–serie).
- Apéndice C. *Ejemplo de una sesión de trabajo con* Retiner: En este apéndice se ofrece un ejemplo completo en forma de tutorial de una sesión de trabajo normal con la herramienta *Retiner*, explorando su espectro de posibilidades de diseño.
- Apéndice D. *Relacción de acrónimos*: Este apéndice muestra el listado de los acrónimos más relevantes empleados en la memoria.
- *Índice de alfabético.* Relacción de los conceptos más importantes junto con los números de página desde donde se les referencia.
- *Bibliografía.* Listado de las referencias bibliográficas junto con el número de página desde donde se les cita.

1.5. Convenciones tipográficas

Con objeto de mejorar la lectura de la presente memoria de tesis doctoral, se han tenido en cuenta las siguientes convenciones tipográficas para enfatizar en cada caso las siguientes supuestos:

- Palabras en lengua distinta a la castellana:

  Very High Speed Hardware Description Language.

- Código fuente (nótese la negrilla para las palabras reservadas del lenguaje):

```
process (SIM_CLOCK) ...
```

- Nombre de un elemento asociado a un código fuente concreto (Ej. nombre de una variable o de un tipo):

`sim_extRAMread`

- Expresión matemática normal:

$$k(x,y) * m(x,y) = \sum_{i=-\frac{[F]}{2}}^{\frac{[F]}{2}} \sum_{j=-\frac{[C]}{2}}^{\frac{[C]}{2}} k(i,j) \cdot m(x-i, y-j) \tag{1.1}$$

- Expresión matemática con vectores (nótese la negrilla para el vector):

$$Retina(\mathbf{x}, t) = S(\mathbf{x}) \cdot T(t) \tag{1.2}$$

- Notas al pie de página:

  Esto es un ejemplo[1].

- Valor numérico:

  Estamos en 2006.

- Énfasis especial en un concepto:

  Esta tesis trata de un desarrollo de *Neuroingeniería*.

- Definición de un concepto:

  VHDL Acrónimo que representa la combinación de *VHSIC* y *HDL*, donde a su vez *VHSIC* es el acrónimo de *Very High Speed Integrated Circuit* y *HDL* es a su vez el acrónimo de *Hardware Description Language*.

[1] de pie de página

Capítulo 2

Diseño automático de sistemas digitales orientados a visión

El capítulo presente hace un análisis de las diferentes estrategias, herramientas y dispositivos que intervienen en el diseño de sistemas de procesamiento visual. También se definen los conceptos y fundamentos computacionales en los que se ha apoyado el trabajo relativo a la presente tesis doctoral. La sección 2.1 introduce el capítulo articulando su contenido. Seguidamente la sección 2.2 analiza el estado del arte, en cuanto a las diferentes opciones plausibles en el diseño de sistemas de procesamiento de visión de alto rendimiento. También se presenta la tecnología FPGA*, con la que se ha llevado a cabo el trabajo doctoral y se definen los conceptos: computación reconfigurable y lenguaje de descripción de hardware. Posteriormente en la sección 2.3, se hace una breve descripción de los distintos métodos y lenguajes con los que diseñar hardware reconfigurable, prestándose más atención a las arquitecturas basadas en FPGAs. La sección 2.4 presenta las alternativas más transcendentes en el diseño y co-diseño hardware/softwareware, prestando especial atención* CodeSimulink*, la herramienta de codiseño que ha definido gran parte del trabajo de esta tesis. Por último, la sección 2.5 concluye el capítulo resumiendo las opciones que se han tomado en el trabajo doctoral.*

2.1. Introducción

Durante las últimas décadas, el desarrollo de las nuevas tecnologías de fabricación de circuitos integrados, junto con el significativo avance en los equipos informáticos, tanto de softwareware como de hardware, ha propiciado la aparición de nuevas arquitecturas electrónicas orientadas al prototipado rápido de sistemas electrónicos y la aparición a su vez de sus herramientas duales de tipo EDA *Electronic Design Automation* que explotan sus numerosas

posibilidades.

Estamos asistiendo a un, cada vez más acelerado, crecimiento en el uso de herramientas destinadas a la síntesis y simulación de sistemas electrónicos partiendo de especificaciones en lenguajes de propósito específico tipo *HDL*. A continuación se analizará el estado actual del diseño de sistemas de procesamiento de visión, definiendo los conceptos y estrategias que se han tomado para desarrollar la neuroprótesis visual objeto de esta tesis doctoral.

2.2. Sistemas de procesamiento visual.

La evolución tecnológica actual demanda de forma cada vez más acusada, diseños de sistemas de procesamiento visual cada vez más potentes, en el sentido de que se requiere un mayor procesamiento de datos (sistemas de inspección visual, sistemas basados en flujo óptico, etc...) en el menor tiempo posible. Entre los sistemas de procesamiento visual que necesitan de un gran rendimiento, encontramos los destinados al procesamiento de imágenes en tiempo–real. Este requisito de cómputo es deseable hoy en día en tareas tales como el reconocimiento biométrico de patrones (reconocimiento de rostros, de patrones de retina, de huellas dactilares, etc...), aplicaciones de seguridad y vigilancia (detección de intrusos), apoyo a diagnósticos médicos, inspección visual en cadenas de producción industrial, etc. En definitiva tareas todas ellas que requieren el procesamiento masivo de datos visuales y una respuesta del sistema en intervalos controlados de tiempo.
A su vez, los requisitos de rendimiento (tanto energético como de procesamiento), fiabilidad, bajo coste y de alta flexibilidad (adaptación a distintos entornos de operación) conducen a elaboraciones en donde cada vez más interviene el diseño de sistemas hardware y sistemas híbridos hardware/softwareware, en detrimento de soluciones basadas en softwareware que muchas veces se evidencian insuficientes.
De esta manera, para diseñar sistemas de procesamiento visual en tiempo real y alto rendimiento (y en general de procesamiento digital de señales) se pueden escoger tres tipos de soluciones tecnológicas (que pueden combinarse entre sí):

- *Soluciones basadas en* ***DSPs***: Estos microprocesadores especializados en procesar señales (*Digital Signal Processor*) ofrecen una gran facilidad de uso (mediante lenguajes estándar adaptados), permiten la reprogramación sobre el mismo circuito y suelen ser una solución bastante económica. Sin embargo, el paradigma de ejecución secuencial en el que están basados, imposibilita la ejecución en tiempo–real de ciertos algoritmos basados en el paralelismo y la replicación de sus subsistemas de cómputo.

- *Soluciones basadas en circuitos* ***ASIC***: Los circuitos integrados de aplicación específica (*Application-Specific Integrated Circuits*) aportan la solución idónea en cuanto al área de silicio ocupada, velocidad de procesamien-

to y bajo consumo. Por otro lado, esta solución presenta los siguientes problemas:

  - ⇝ Los tiempos medios de diseño suelen ser excesivos (hoy en día la variable *time-to-market*[1] es un parámetro de suma importancia).
  - ⇝ La no-reprogramabilidad de estos dispositivos aumenta el riesgo de diseño de forma considerable, a la vez que dificulta la adaptación del sistema a nuevos esquemas de cómputo, mejoras, nuevos parámetros, etc.
  - ⇝ Sólo grandes tiradas de producción justifican el gran desembolso que lleva consigo su fabricación. Esto hace que no sea el dispositivo electrónico óptimo para probar prototipos de sistemas.

- *Soluciones basadas en PLDs*: Los modernos dispositivos lógicos programables o *PLDs* (*Programmable Logic Device*), entre los que se destacan los dispositivos de matrices de puertas programables por campo o *FPGAs* (*Field Programmable Gate Array*) ofrecen una solución intermedia frente a las anteriores en cuanto a rendimiento, prestaciones y consumo, aportando además la flexibilidad de softwareware.

Ligado intimamente a la tecnología *FPGA* y respondiendo a la necesidad creciente de particularización ágil de modelos y adaptación al entorno operativo que exige el diseño moderno de sistemas hardware/softwareware, aparece el concepto de *computación reconfigurable* que se precisa a continuación:

Computación reconfigurable[2] : Este concepto, introducido hace cuatro décadas, tiene actualmente una interpretación bastante amplia. En cuanto a arquitecturas de implementación, abarca soluciones tan variadas como las redes de procesadores con conexiones configurables, las plataformas hardware/softwareware que combinan procesadores estándar y coprocesadores reconfigurables y los *PLDs*, fundamentalmente la tecnología *FPGA*. El nexo común que unifica las soluciones anteriores y define el concepto de computación reconfigurable es la cualidad de ciertos sistemas hardware de poder ser redefinidos o reprogramados total o parcialmente.

Los requisitos propios de la presente tesis doctoral, han encaminado el diseño de los prototipos de la neuroprótesis que nos ocupa hacia una implementación en hardware reconfigurable utilizando la tecnología *FPGA* que se presenta a continuación.

2.2.1. La tecnología FPGA.

De todas las soluciones de computación reconfigurable, sin duda, la que más ha evolucionado es el la plataforma *FPGA*. Hoy en día se pueden encontrar chips *FPGA* que integran elementos de cómputo de granularidad muy diversa,

[1] Tiempo que transcurre desde que se concibe una idea hasta que ésta llega al mercado.

desde pequeños núcleos de cálculo hasta procesadores y memorias embebidas. Aplicándose todavía la *Ley de Moore*, estos dispositivos integran cada vez más transistores en menos espacio lo que maximiza las posibilidades de realización de algoritmos cada vez más complejos y abarata costes.

Acrónimo de *Field-Programmable Gate Array*, las *FPGAs* son dispositivos que contienen una red de puertas lógicas y otros elementos de cómputo más complejos cuyas conexiones pueden ser programables por el usuario. En la actualidad se pueden integrar millones de dichas puertas lógicas en cada chip.

La arquitectura de una *FPGA* consiste en muchas celdas lógicas pequeñas distribuidas regularmente por su superficie. Cada celda lógica contiene 1 ó 2 registros y pequeñas *LUTs* (*Look–Up Table*) que son capaces de implementar funciones lógicas sencillas. Las celdas lógicas suelen contener multiplexores para conectar las *LUTs* con registros. Para interconectar las celdas lógicas se utilizan canales de interconexión regularmente distribuidos a lo largo de toda la superficie. No obstante, pueden existir varios niveles de interconexión entre las primitivas de cómputo que se integran en el chip según la cercanía de éstos entre sí. La velocidad de procesamiento de las *FPGAs* es generalmente menor que la que se puede obtener con un *ASIC* y suelen consumir más energía. Sin embargo, y tal como se ha comentado, presentan ventajas muy provechosas a causa de su reprogramabilidad (incluso *en caliente*), que permite corregir errores de diseño de forma cómoda y muy eficiente. Otras de las características más provechosas de las *FPGAs* es la posibilidad de *transportar* el diseño a un circuito *ASIC* gracias a que usualmente la especificación de sistemas hardware en *FPGA* viene descrito mediante uno ovarios lenguajes de descripción de hardware, concepto que se define a continuación.

HDL Acrónimo de *hardware Design Language*. Se trata de un conjunto de expresiones en texto que expresan el comportamiento temporal y la estructura espacial de un sistema electrónico. Una sintaxis de un *HDL* incluirá notaciones explícitas para expresar la concurrencia y la planificación temporal de un circuito, atributos primordiales de todo hardware. Generalmente un *HDL* permite definir un modelo para simulación o fuertemente comportamental (y por tanto sujeto a las especificaciones temporales y de comportamiento que describa el ingeniero) y otro para síntesis, susceptible de ser llevado a una cierta arquitectura electrónica para su implementación real. El modelo para síntesis puede también simularse, aunque las restricciones estarán impuesta por las características electrónicas del dispositivo físico donde se quiere implementar el diseño. Algunas herramientas permiten sintetizar un subconjunto fuertemente comportamental de un cierto *HDL* .

El binomio computación reconfigurable–*HDL*, está siendo utilizado hoy en día de forma satisfactoria en campos tan competitivos y dependientes del *time-to-market* como el diseño de pequeñas unidades de control dentro de los más modernos procesadores (los *Pentium* 4 por ejemplo) y el diseño de *DSPs* para teléfonos móviles.

2.3. Lenguajes de descripción de hardware y Cosimulación

Como se comentaba en la sección 2.2, la inercia del mercado y la industria electrónica ha dado paso a las nuevas tecnologías de dispositivos lógicos programables que se comentan en él. La entrada de especificación del diseño electrónico ha tenido inexorablemente que cambiar desde el puro diseño de máscaras de los circuitos *Full Custom*, hasta los modernos lenguajes específicos orientados a hardware que se comentan más adelante[3]. Al aparecer en el mercado las nuevas herramientas *EDA*, que empezaban a integrar de forma paulatina y dentro del mismo entorno, las tareas de descripción, síntesis y simulación del sistema electrónico, se hizo necesaria la idea de disponer de una forma estándar y portable entre dichas herramientas, de describir el sistema electrónico. Nacieron así los lenguajes orientados a hardware o *HDLs*.

Las principales características de las herramientas EDA más importantes (*Xilinx ISE*, *Altera Quartus*, *Altera MaxPlus+II*, *Leonardo Spectrum*, Synopsys...) que intervienen en el diseño de circuitos son:

- Integración de uno o más lenguajes de descripción de hardware.
- Posibilidad de edición de diagramas esquemáticos.
- Grafos y Diagramas de Flujo. Edición de máquinas de estados finitos.
- Simulación lógica funcional o comportamental.
- Simulación lógica post–síntesis (con retardos reales).

Definiendo nivel de un lenguaje de programación como el grado de abstracción que brinde dicho lenguaje frente a la plataforma hacia la que se está diseñando el programa, se puede esbozar la siguiente tabla comparativa 2.1.

Por simplicidad, he añadido en la tabla los lenguajes softwareware y hardware más importantes inspirados en el lenguaje *C*. No en vano, son los que más éxito tienen dentro de la industria y en el ámbito universitario. Así, nos encontramos lenguajes como *C++*, *C#*, *Java* (*JHDL*), *SystemVerilog*, *SystemC*, *HandelC* todos ellos basados en *C*.

Hago a continuación una breve reseña de los *HDL* más importantes que podemos encontrar hoy en día:

***VHDL*:** Acrónimo de *VHSIC HDL*, siendo *VHSIC* el acrónimo de *Very High Speed Integrated Circuit*, se trata del *HDL* más extendido y de mayor éxito hasta la fecha. Los sintetizadores lógicos han apoyado de forma especial a este lenguaje desde sus comienzos, y presentan las optimizaciones más avanzadas y estudiadas frente a los demás *HDLs*.

[3]No quiere esto decir que el diseño *Full Custom* esté en desuso, pero sí que ha perdido popularidad en ciertas parcelas del diseño electrónico, como por ejemplo las que priman el *time–to–market* o el desarrollo de prototipos funcionales.

Tabla 2.1: Comparativa de lenguajes softwareware y hardware atendiendo al nivel de abstracción.

SOFTWARE	***HARDWARE***	***Nivel***
Lenguajes ensambladores (*NASM*, *MASM*, ...)	Lenguajes tipo *netlist* (*EDIF*, *Xilinx XNF*, ...)	bajo
Lenguajes para el diseño se sistemas (*C*)	*HDL* Estructural (*VHDL* y *Verilog* estructural)	medio–bajo
Lenguajes de propósito general (*C*, *C++*, ...)	HDL RTL { Flujo de datos / Algorítmico (*VHDL* y *Verilog*)	medio–alto
Lenguajes avanzados (*C++* , *C#*, *Java*, ...)	*HDL comportamental* (*VHDL* y *Verilog* comportamental, *System C*, *SystemVerilog*, *HandelC*, ...)	Alto

VHDL heredó la sintáxis del lenguaje softwareware *Ada*, muy conocido en ámbientes donde se requiere una alta seguridad y control en el código generado, trabajar con sistemas en tiempo real y un completo manejo de errores, siendo muy utilizado en misiones espaciales, y por empresas militares. Concretamente, *VHDL* extendió este lenguaje para poder describir mejor la concurrencia de eventos en los sistemas electrónicos.

***Verilog*:** *Verilog* (a veces también *Verilog HDL*) ha sido siempre el *HDL* que ha rivalizado en popularidad con *VHDL*. Su menor éxito ha sido atribuido a la lentitud en el proceso de estandarización por parte de la empresa (*Cadence*) que tenía sus derechos. Al igual que *VHDL*, soporta la especificación, simulación y síntesis de circuitos digitales, analógicos y mixtos, al igual que varios niveles de abstracción en la definición de los distintos circuitos. *Verilog* está inspirado en lenguaje *C* de los años 70, heredando de este un preprocesador parecido y las sentencias (if, while, etc.) más comunes. Hoy en día la mayoría de herramientas de síntesis lógica más conocidas (*Xilinx ISE*, *Leonardo Spectrum*, ...) añaden soporte tanto para *VHDL* como para *Verilog*.

***Handel-C*:** Este *HDL* nació en la Universidad de Cambridge y su rápida popularidad en entornos universitarios lo llevó, ayudado por el *spin-off* que originó y que posteriormente se convirtió en *Celoxica Ltd.*, a hacerse un hueco dentro de los lenguajes *HDL* de más alto nivel. Su éxito inicial vino auspiciado tanto por la herramienta (*Celoxica DK*) que integraba un motor de simulación y síntesis dentro de un entorno amigable, como por la enorme similitud y compatibilidad (en simulación) con el lenguaje *C*. La estrategia inteligente consistió en desarrollar un *HDL* pseudo-C, con sentencias específicas para manejar el paralelismo inherente a todo sistema

digital y hacer que para la simulación del diseño, el mismo código pudiera ser compilado y ejecutado con la ayuda de un compilador estándar (como *GNU gcc*).

2.4. Entornos de diseño y co–diseño de hardware

La creciente demanda de sistemas empotrados (*embedded systems*) ha originado un nueva estrategia de diseño de sistemas híbridos hardware/softwareware usualmente referida por *codiseño hardware/softwareware* o sencillamente *codiseño*.

Codiseño [2]: Estrategia de diseño de sistemas hardware/softwareware que engloba los siguientes ítems:

- Estudio la interacción entre los modelos de diseño hardware y de diseño softwareware.
- Exploración detenida del espacio de diseño para definir una partición hardware/software óptima.
- Diseño de una interfaz hardware/softwareware flexible.
- La posibilidad de simulación conjunta softwareware/hardware, también conocida como *co-simulación*.

Uno de los efectos del codiseño en el desarrollo de sistemas de computación es la relajación de las fronteras entre hardware y softwareware. Cuando en un sistema *Hw/Sw* son importantes ambos aspectos así como la interacción o interfaz entre cada una de las partes, el codiseño *Hw/Sw* se erige como una de las estrategias de diseño más rentables. En la actualidad se pueden encontrar numerosas herramiendas de tipo *CAD* (*Computer Aided Design*) que permiten el codiseño *Hw/Sw* facilitando el proceso de diseño de sistemas *Hw/Sw*.
A continuación se describen las herramientas actuales de *codiseño Hw/Sw* más notables, haciendo especial énfasis en el entorno *CodeSimulink*, que ha sido utilizado y modificado de forma conveniente en el desarrollo de la presente tesis:

2.4.1. El entorno de codiseño *CodeSimulink*

Introducción

CodeSimulink o también *SMT6040* (nombre con el que se comercializa el producto por la empresa *Sundance*) es una potente herramienta capaz de generar código *VHDL* a partir de diagramas *Simulink*. Originalmente fue diseñada como herramienta de desarrollo para las placas basadas en *FPGA* de *Sundance*, como *SMT355*, *SMT358*, *SMT398*, etc., aunque en la actualidad, se ha ampliado el conjunto de *FPGAs* y de placas basadas en *FPGA* a las que se da soporte de forma nativa.
Diseñar subsistemas hardware con *CodeSimulink* es tan fácil como desarrollar

y simular un diagrama *Simulink*. Esto hace que usuarios sin mucha experiencia en el diseño de hardware, puedan desarrollar sistemas hardware y hardware/softwareware de alta eficacia y bajo coste que además sean portables entre distintas arquitecturas de tipo *FPGA*. Mediante *CodeSimulink* los sistemas hardware se diseñan y simulan por medio del entorno *CodeSimulink*, que ha sido enriquecido con un conjunto de nuevos bloques[4] capaces de describir sistemas hardware. Actualmente se da soporte hardware mediante estos bloques, a la práctica totalidad de los bloques *Simulink*, que tienen así un *dual* con nuevos parámetros *hardware* asociados. El modelo de procesamiento que implementa *CodeSimulink* está pensado para sistemas orientados al flujo de datos o *data–flow*. No obstante también es posible implementar bloques orientados a control y que se integren en un esquema *CodeSimulink* gracias a su protocolo (que veremos más adelante).

CodeSimulink cierra el ciclo del diseño de hardware, es decir, está concebido para:

- *Modelar* sistemas híbridos hardware/softwareware
- *Simular* el sistema con las restricciones hardware.
- *Depurar* el modelo con la ayuda de *Simulink* y *Matlab*.
- *Sintonizar* o ajustar los parámetros del modelo.
- *Optimizar* el diseño buscando los mejores valores de los parámetros.
- *Compilar* o conectar con la herramienta de síntesis lógica seleccionada.
- *Configurar* la *FPGA* y/o *Programar* o *Ejecutar* el modelo completo con la ayuda de aplicaciones externas.

sistemas *HW* y *HW/SW* de forma rápida y eficaz, lo que lo convierte en una herramienta altamente competitiva dentro del mercado de prototipado rápido de sistemas digitales, en el que el parámetro *time–to–market* cobra meridiana importancia.

Entre las ventajas que se pueden encontrar en *CodeSimulink* se acentúan las siguientes:

- Aproximación directa y muy intuitiva al diseño hardware de sistemas digitales.
- Compatibilidad con los bloques existentes de *Simulink*
- Reutilización directa de los distintos componentes *CodeSimulink* así como de las estructuras y modelos definidas por el usuario.

[4] *bloques CodeSimulink*

- Simulación integrada en el entorno *Simulink*, que permite realizar una rápida cosimulación exacta[5] del sistema, interaccionando cuando estén presentes, con los subsistemas softwareware y dispositivos hardware involucrados en el sistema.

- Fuerte integración con *Matlab*, que permite desarrollar *scripts* y funciones en lenguaje *Matlab*, que interaccionen con nuestro sistema para realizar un ajuste automático de los parámetros hardware más relevantes (resolución, frecuencia de muestreso, etc.), permitiendo una optimización ágil y potente del sistema.

- Compatibilidad con las siguientes herramientas comerciales de síntesis lógica:

  ⇝ *Mentor Graphics Leonardo Spectrum Level 3 2000* o posterior.

  ⇝ Suite *Xilinx ISE Foundation 6.2*, integrada con la herramienta de síntesis *Xilinx XST* (*Xilinx Synthesis Tool*).

  ⇝ *Altera Quartus II*.

- Compatibilidad con las siguientes herramientas comerciales de *Place*&*Route* para *FPGAs*:

  ⇝ Suite *Xilinx ISE Foundation 6.2* o posterior.

  ⇝ *Altera Max–Plus II* o posterior.

  ⇝ *Quartus II* o posterior.

Esquema de procesamiento de *CodeSimulink*.

Una de las características más notables de *CodeSimulink* es el empleo de un *protocolo distribuido*, en adelante *protocolo CodeSimulink* diseñado para:

- sincronizar los diferentes módulos hardware,

- planificar el procesamiento de diferentes tipos de datos (matrices, vectores, escalares),

- y asegurar la fiabilidad de los datos que se procesan.

El empleo de un protocolo distribuido, en lugar de uno *centralizado*, es una de las características más diferenciadoras de *CodeSimulink* comparado con otras herramientas análogas como *Xilinx System Generator* o *Altera DSP Builder*.

De otra parte, para entender este protocolo, conviene hablar antes de los tipos de datos soportados por *CodeSimulink*, característica crucial a la hora de compararlo con otras herramientas similares.

[5]Simulación que tiene en cuenta las restricciones hardware del sistema simulado.

Tipos de datos soportados por *CodeSimulink*:

Mientras que *Simulink* no hace una distinción muy fuerte de los tipos de datos normales que soporta, es decir, escalares, vectores y matrices, la diferencia entre éstos se hace muy notable a la hora de representar y procesar esta información desde el punto de vista del *hardware*. Como se argumentará a continuación, la elección de uno u otro puede tener una incidencia clara en el área ocupada o la velocidad del sistema final sintetizado.

CodeSimulink soporta los siguientes tipos de datos (ver figura 2.1):

- *Escalares* o **`scalar`**: Tipo de dato que representa un solo valor, que usualmente suele estar muestreado en el tiempo a un frecuencia constante. Mientras que en *Simulink*, una matriz o vector de dimensiones 1×1 se comporta como un escalar, *CodeSimulink* hace una clara distinción entre ellos, haciendo necesario el uso del bloque *Reshape* para convertir un tipo en otro y vice versa.

- *Vectores* o **`vector`**: Este tipo de dato representa una serie de N valores independientes, debidamente ordenados que son procesados como un todo. También suelen ser muestreados a una frecuencia constante, es decir N muestras cada período. Este tipo de dato se comporta como una matriz $N \times 1$ en *Simulink*, mientras que para *CodeSimulink* son tipos distintos y al igual que el caso anterior hay que utilizar el bloque *Reshape* para proceder con la conversión de un tipo en otro. Al contrario que los vectores N–dimensionales en *Simulink*, que pueden ser visualizados horizontal y verticalmente según activemos o no un determinado bit selector[6], *CodeSimulink* mantendrá la *dirección* del vector en todos sus bloques.

  Cada componente del vector es transferido y procesado de forma secuencial, cada ciclo de reloj y comenzando por el primero. De este modo si el vector tiene por dimensiones $N \times 1$, se necesitarán N ciclos de reloj para que sea transferido y/o procesado. Consecuentemente el período de muestreo no puede ser menor que N veces el período de reloj, o lo que es igual, la frecuencia de reloj del sistema debe ser menor que N veces la frecuencia de muestreo.

- *Vectores paralelos* o **`parallelVector`**: Se trata de un tipo `vector` con la peculiaridad de que todas sus componentes se transfieren y/o procesan en parelelo. La complejidad del *hardware* involucrado para procesar estos tipos de datos es mayor que para los vectores sencillos, pero se procesa y/o transfiere un vector por ciclo de reloj.

- *Matrices* o **`matrix`**: Una matriz representa un numero independiente de valores, ordenados en forma de estructura rectangular de dimensiones $M \times N$ (M filas por N columnas) que se procesa como una sola entidad, tanto por *Simulink* como por *CodeSimulink*. Como ya comentamos en los

[6] Los bloques que producen salida en *Simulink* tienen la opción *Interpret vector parameters as 1-D* que activada, hace que las matrices de dimensiones $1 \times N$ y $N \times 1$ se transformen en el tipo vector.

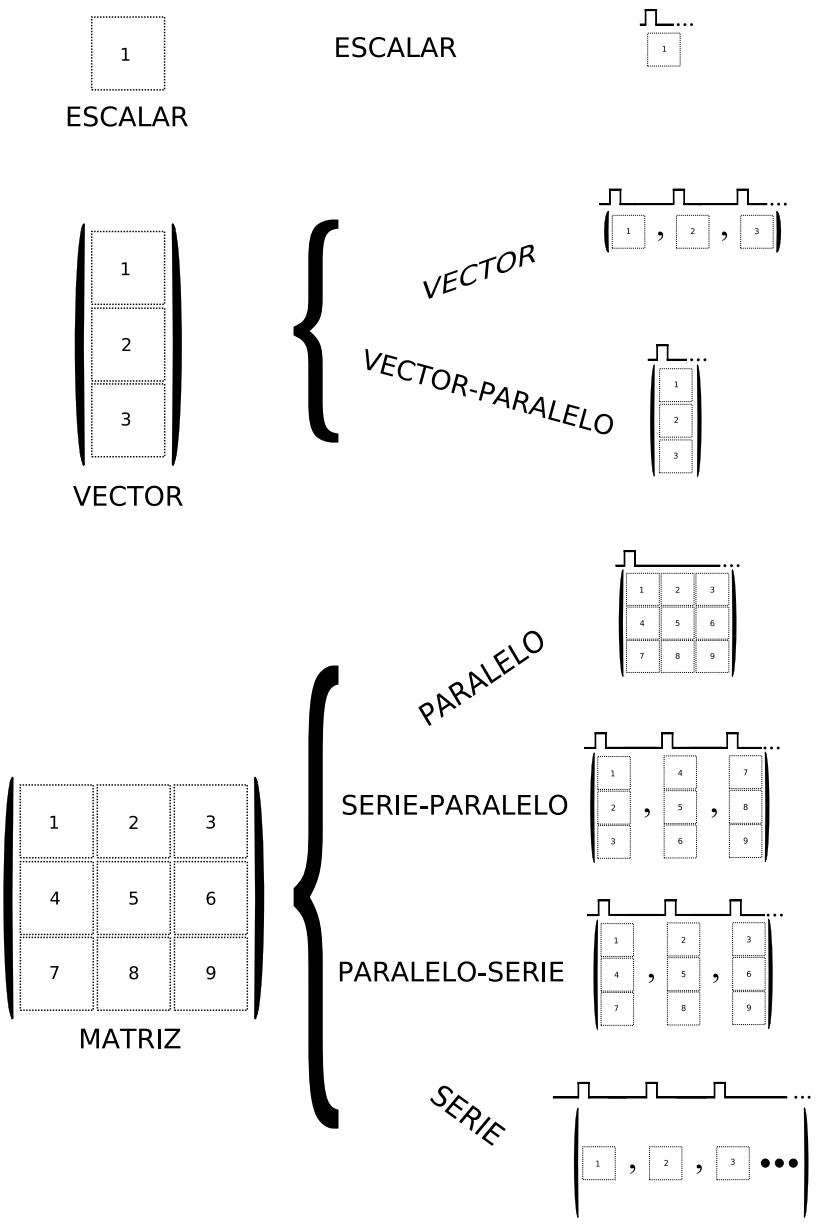

Figura 2.1: Esquema de procesamiento llevado a cabo por los diferentes tipos de datos soportados por *CodeSimulink* (escalares, vectores y matrices), en el que se muestra las posibles formas de procesamiento de que se dispone.

tipos anteriores, los tipos `matrix` de *CodeSimulink* son incompatibles con los tipos `vector`, debiendose utilizar el bloque *Reshape* para convertir un tipo en otro, respectivamente. Los datos de tipo `matrix` suelen ser muestreados a una frecuencia constante de $M \cdot N$ muestras cada período de muestreo. El período de muestreo debe ser mayor que $M \cdot N$ veces el período de reloj del sistema. Las componentes de un dato de tipo `matrix` se procesan y/o transfieren de forma secuencial cada período de reloj, de forma que se toma primero la primera componente de la primera fila, hasta la última componente de dicha fila, para cada una de las filas de la matriz.

- *Matrices Serie–Paralelo* o **`serialParallelmatrix`**: Este nuevo tipo aportado por *CodeSimulink* es análogo al tipo `matrix` salvo por el hecho de que dada una matriz $M \times N$, las M componenes de cada columna se transfieren y/o procesan de forma secuencial, mientras que las N componentes de una fila se transfieren y/o procesan en paralelo. La complejidad del *hardware* es M veces mayor que en el caso `matrix`, pero a cambio M ciclos de reloj son suficientes para transferir y/o procesar toda una matriz.

- *Matrices Paralelo–Serie* o **`parallelSerialmatrix`**: Idénticas a las anteriores salvo que en este caso se procesan N componentes de una fila de forma secuencial y las M componentes de una columna en paralelo. En este caso se necesitan N ciclos de reloj para transferir y/o procesar toda la matriz y la complejidad del *hardware* es N veces mayor que con el tipo `matrix`.

- *Matrices Paralelo–Paralelo* o **`parallelParallelmatrix`**: Se trata de un tipo `matrix` en el que todos los $M \cdot N$ componentes son transferidos y/o procesados en paralelo en sólo un ciclo de reloj. La complejidad del *hardware* es por tanto $M \cdot N$ veces mayor que el caso `matrix`.

Representación de las señales en *CodeSimulink*

Todas las señales del entorno *CodeSimulink* se definen por medio de una serie de parámetros que están presentes tanto en la interfaz de parámetros de cada uno de los bloques de *CodeSimulink*, como en el código *VHDL* generado. Las señales (*signals*) *CodeSimulink* se nombran de tal forma que las entradas a cada uno de los bloques comiencen por `A`, seguido del cardinal indicativo de la entrada de que se trate. De esta forma si nuestro bloque tiene 3 entradas, se tendrán que definir las entradas $A1$, $A2$ y $A3$. Las salidas se nombran análogamente tomando siempre `Y` como la primera letra.

Un bloque *CodeSimulink* puede tener señales de datos de entrada y de salida, dependiendo si se trata de un bloque inicial, intermedio o final. De forma similar, es obligatorio que contenga señales del protocolo *CodeSimulink* de entrada y/o salida, según convenga con el tipo de bloque. La figura 2.2 muestra el esquema general de un bloque *CodeSimulink*, donde se contemplan las señales

destinadas al procesamiento principal y las correspondientes al protocolo *CodeSimulink*.

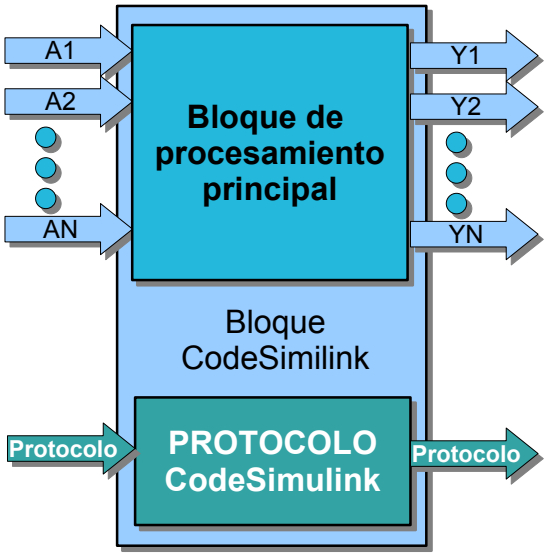

Figura 2.2: Esquema de las salidas y entradas de un bloque *CodeSimulink*

Para caracterizar un módulo *CodeSimulink* se utilizan los siguientes parámetros:

- Longitud de datos o *Data width*: Es el entero positivo que indica el número total de bits que se necesitan para codificar una señal. En la implementación `parallelParallelmatrix`, indica el número de líneas que deben usarse para transferir la señal, mientras que para la implementación `matrix` determina el número de ciclos usados para transferir la señal. En la implementación `parallelSerialMatrix`, indica el número de bits que se transmiten en paralelo en cada ciclo de reloj, es decir, si se tiene una matriz de octetos de dimensiones 4×3, la longitud de datos sería $4 \cdot 8 = 32$ bits. El código *VHDL* generado marcará este parámetro como `sim_DataWidth_Y1`, `sim_DataWidth_Y2`, ...o `sim_DataWidth_A1`, `sim_DataWidth_A2`, ...según se trate de salidas o entradas respectivamente.
- Posición del punto binario o *Binary point position*: Es un entero (positivo o negativo) que indica el número de bits que se emplean para codificar la mantisa o parte significativa del dato. El código *VHDL* creará parámetros con el prefijo `sim_BinaryPoint_+nombre del puerto`.
- Representación de la señal o *Signal representation*: Codificado en *VHDL* por el parámetro `sim_SigRep_+nombre del puerto`, se trata del modo de codificación de la señal en *CodeSimulink*. Es posible escoger entre:

  - ⇝ Sin signo o `unsigned`: Enteros positivos en punto fijo o *fixed–point* y codificados en binario.
  - ⇝ Con signo o `signed`: Enteros en complemento a dos en punto fijo y codificados en binario.
  - ⇝ Módulo y signo o `signModulus`: Enteros en punto fijo, codificados en binarios siguiendo la forma *signo* + *módulo*.
  - ⇝ Real en coma flotante o `float`: Real en punto flotante codificado según el estándar *IEEE–754* ampliado para soportar cualquier valor de *longitud de datos*.
  - ⇝ Real en coma flotante normalizado o `normFloat`: Que representa sólo el valor normalizado del estándar *IEEE–754*. El rango de valores a representar se reduce, pero la complejidad del hardware involucrado se reduce significativamente.
  - ⇝ Representación heredada del módulo anterior o `backPropagated`: Dada una salida `YN`, con $N = 1, 2, \ldots$, se hereda la representación de la entrada `AN` correspondiente.

- Computación de señales no representables o *Non representability management*: Parámetro que define qué hacer en caso de que el valor de la salida del bloque esté fuera del rango permitido, quedando dicho rango definido por los parámetros *Data width*, *Binary point position* y *Signal representation*. El parámetro *VHDL* generado toma la forma `sim_Overflow_`+`nombre del puerto de salida`.
  Es posible elegir una de las opciones siguiente:
  - ⇝ *Saturación* o `saturarion`: Que satura la salida al máximo valor representable, bien sea negativo o positivo.
  - ⇝ *Ciclado* o `wraparound`: Que cicla el valor de la señal, de forma que después del máximo valor permitido se comienza por el mínimo permitido y viceversa. Esta opción sólo se aplica a las representaciones con y sin signo.
  - ⇝ Herencia del módulo anterior o `Back-propagated`: Para heredar el valor del bloque de entrada correspondiente.

- Método de redondeo o `Rounding method`: Que designa la actuación en caso de que el valor a codificar no se pueda alcanzar de forma exacta con la resolución que haya definido. El parámetro *VHDL* generado toma la forma `sim_Rounding_`+`nombre del puerto de salida`.
  Se puede escoger entre las opciones siguientes:
  - ⇝ `ceil`: Para redondear al valor representable más pequeño, no menor que el original.
  - ⇝ `floor`: Para redondear al valor representable más grande, no mayor que el original.

- ⇝ `round`: Redondea al valor representable más cercano, pudiendo ser mayor o menor que el original.
- ⇝ `fix`: Redondea al primer valor representable más cercano hacia cero.
- ⇝ *Herencia de módulo anterior* o `Back-propagated`: Para tomar la misma opción de redondeo que el puerto de entrada correspondiente.

El protocolo de *CodeSimulink*

Como se ha comentado en la sección 2.4.1, el protocolo de *CodeSimulink* es un protocolo distribuido, que sincroniza los distintos módulos hardware, planifica el procesamiento de los distintos tipos de señales que puede manejar *CodeSimulink* y asegura la fiabilidad de los datos que se procesan. En reglas generales este protocolo asegura que sólo se procesan las señales de entrada que estén listas (`ready`) para ello, y se transfiere la salida procesada sólo cuando el o los siguientes bloques sucesivos que constituyen el (*fan–out*) o árbol de conexiones salientes, esté listo para recibir una nueva señal.
El protocolo de *CodeSimulink* se construye a través de una serie de señales de entrada y de salida que comunican los bloques que están unidos entre sí. Cada entrada $A1, A2, \ldots$ y cada salida $Y1, Y2, \ldots$ tiene asociados un conjunto de señales de protocolo, que se dividen en:

- **Señales del protocolo de entrada**, son las señales del protocolo que arbitran el procesamiento y la transferencia de datos entre un bloque dado y el/los precedentes. Este conjunto está definido por las señales siguientes:
  - ⇝ `DIGIO`: Es el puerto de entrada de los datos que debe procesar el bloque en cuestión.
  - ⇝ `IN_VAL`: Los datos de entrada se toman como válidos por medio de esta señal definida como un vector de 4 bits, que define a su vez las señales siguientes:
    - ↪ `IN_VAL0`: Señal también llamada `VALID` o sencillamente `VAL`, que vale 1 si y sólo si los datos que se presentan a la entrada son válidos y pueden ser utilizados por el bloque de forma confiable en el siguiente ciclo de reloj. Si no se asigna nigún valor para esta señal, se ancla a 1, lo que significa que los datos son siempre válidos, tal y como correspondería a un protocolo de transferencia continua.
    - ↪ `IN_VAL1`: También llamada `EOF` (del inglés *End Of File*), se trata de una señal que vale 1 cuando llega a la entrada el último elemento de un vector (en el caso de que la señal sea de tipo vector) o el último elemento de una fila (en el caso de una matriz).
    - ↪ `IN_VAL2`: Esta señal, también llamada `EOM` (del inglés *End Of Matrix*) se pone a 1 cuando llega a la entrada el último elemento de la última fila de una matriz.

  - ↪ IN_VAL3: La cuarta de las señales de IN_VAL se define también por GLOBAL_READY o simplemente GRD. Si sólo hay un bloque fuente, es decir, si el bloque en cuestión sólo recibe entrada de otro bloque *CodeSimulink*, entonces puede dejarse sin asignar o se puede anclar su valor a 1. Por el contrario, en caso de tener la salida de varios bloques como entrada del que nos ocupa, esta señal debe ser la función *Y lógica* de todas las señales IN_RDY (que se define a continuación) de todos los bloques de entrada.

- ⇝ IN_RDY: También llamada READY o RDY, esta señal de salida de un sólo bit indica que el bloque se encuentra preparado para recibir un nuevo dato. Cuando vale 1 el bloque estará listo para leer un nuevo dato en el siguiente ciclo de reloj siempre y cuando VALID $= 1$ y GLOBAL_READY $= 1$. Si IN_RDY $= 0$, el bloque no estará listo para recibir, por lo que el puerto de entrada (DIGIO) y las señales VALID permanecerán inalteradas hasta que IN_RDY $= 1$.
- ⇝ OUT_CLOCK: Señal de reloj. Opcionalmente es posible conectar a este puerto de salida la señal de reloj interna SIM_CLOCK.
- ⇝ OUT_CLEAR: Señal de reset síncrono. Opcionalmente es posible conectar a este puerto de salida la señal interna de reset SIM_CLEAR

- **Señales del protocolo de salida**,son las señales del protocolo que arbitran el procesamiento y la transferencia de datos entre un bloque dado y el/los precedentes. Este conjunto está definido por las señales siguientes:
  - ⇝ DIGIO: Es el puerto de salida de los datos que se han procesado.
  - ⇝ OUT_VAL: Los datos de salida se toman como válidos por medio de esta señal definida como un vector de 4 bits, que define a su vez las señales siguientes:
    - ↪ OUT_VAL0: Señal también llamada VALID o sencillamente VAL que vale 1 si y sólo si los datos a la salida son válidos y pueden ser utilizados por el bloque siguiente de forma confiable en el siguiente ciclo de reloj.
    - ↪ OUT_VAL1: También llamada EOF (del inglés *End Of File*), se trata de una señal que vale 1 cuando llega a la salida el último elemento de un vector (en el caso de que la señal sea de tipo vector) o el último elemento de una fila (en el caso de una matriz).
    - ↪ OUT_VAL2: Esta señal, también llamada EOM (del inglés *End Of Matrix*) se pone a 1 cuando llega a la salida el último elemento de la última fila de una matriz.
    - ↪ OUT_VAL3: La cuarta de las señales de OUT_VAL se define también por GLOBAL_READY o simplemente GRD. Esta señal es una copia exacta de OUT_RDY (que se define a continuación). No es necesaria su asignación.

- ⇝ `OUT_RDY`: También llamada `READY` o `RDY`, esta señal de entrada de un sólo bit indica que el bloque de destino se encuentra preparado para recibir un nuevo dato. Cuando vale 1 el bloque de destino estará listo para leer un nuevo dato en el siguiente ciclo de reloj siempre y cuando `VALID` = 1 y `GLOBAL_READY` = 1. Si `OUT_RDY` = 0, el bloque en cuestión no podrá enviar ningún dato, por lo que el puerto de salida (`DIGIO`) y las señales `VALID` permanecerán inalteradas hasta que `OUT_RDY` = 1.
- ⇝ `OUT_CLOCK`: Señal de reloj. Opcionalmente es posible conectar a este puerto de salida la señal de reloj interna `SIM_CLOCK`.
- ⇝ `OUT_CLEAR`: Señal de reset síncrono. Opcionalmente es posible conectar a este puerto de salida la señal interna de reset `SIM_CLEAR`

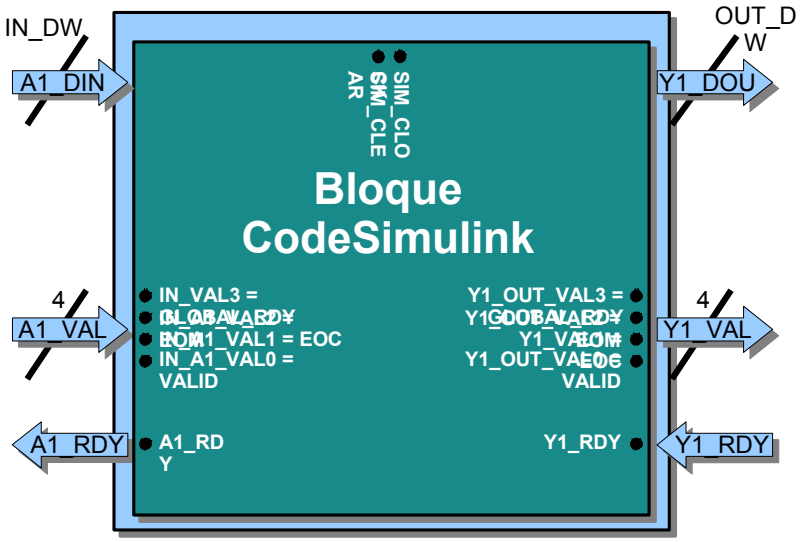

Figura 2.3: Esquema completo de las señales que maneja un bloque *CodeSimulink*, en el que se resaltan las señales principales del protocolo distribuido de *CodeSimulink*.

La figura 2.3 muestra el esquema de las señales (datos + protocolo), que afectan a un bloque *CodeSimulink*. Las señales `SIM_CLEAR` y `SIM_CLOCK` corresponden a las entradas internas de *reset* y reloj del bloque. Opcionalmente pueden asignarse a los puertos `OUT_CLEAR` y `OUT_CLOCK`, que han sido eliminados por simplicidad. La entidad *VHDL* que engloba una celda de protocolo *CodeSimulink* contiene 21 puertos, 4 de entrada y 17 salida. Con objeto de ilustrar la complejidad y potencia del protocolo, muestro a continuación una pieza de código del componente principal de dicho protocolo, en la que se puede observar la definición de los puertos de entrada y de salida del protocolo:

```
COMPONENT sim_prot_synchpar
  GENERIC (
   SIM_PIPELINE : INTEGER := -1;
    SIM_INPUTS_NUMBER : NATURAL := 1;
    A : SIM_SIGNAL_ATTRIBUTES := SIM_DEFAULT_ATTRIBUTES;
    Y1 : SIM_SIGNAL_ATTRIBUTES := SIM_DEFAULT_ATTRIBUTES
  );

  PORT(
    IN_VAL_VECTOR : IN SIM_SIGVAL_SYNCHPAR_VECTOR
                   (0 to SIM_INPUTS_NUMBER-1);
    IN_RDY : OUT STD_LOGIC_VECTOR
             (0 to SIM_INPUTS_NUMBER-1);
    Y1_VAL : OUT sim_sigval_synchpar;
    Y1_RDY : IN STD_LOGIC := '1';
    RECEIVING : OUT STD_LOGIC;
    TRANSMITTING : OUT STD_LOGIC;
    VALID : OUT STD_LOGIC;
    NOMOREVALID : OUT STD_LOGIC;
    P_EN : OUT STD_LOGIC;
    E_IV : OUT STD_LOGIC;
    E_IM : OUT STD_LOGIC;
    E_IS : OUT STD_LOGIC;
    READY: IN  STD_LOGIC := '1';
    USED : OUT STD_LOGIC;
    NEXT_OUTDATA : OUT STD_LOGIC;
    OUTVALID : OUT STD_LOGIC;
    E_OV : OUT STD_LOGIC;
    E_OM : OUT STD_LOGIC;
    E_OS : OUT STD_LOGIC;
    R_EN : OUT STD_LOGIC;
    SIM_CLEAR, SIM_CLOCK : IN STD_LOGIC
  );
END COMPONENT;
```

Existen versiones adaptadas u optimizadas del protocolo para el caso de que se quieran definir componentes que no precisen de todas las señales de dicho protocolo, o que se comporten como *fuentes* o *sumideros* de datos. Por ejemplo, un módulo para acceder a memoria *RAM* externa cuya entrada no esté conectada con ningún bloque *CodeSimulink*, sería modelado como un bloque con una salida determinada pero sin ninguna entrada, es decir, una fuente de datos de la que no sería necesario el arbitraje o *handshaking* con ningún bloque precedente, pudiéndose recortar el protocolor en este sentido. Este es el caso de algunos de los bloques que se han desarrollado durante este trabajo doctoral (consultese la sección 6.3.3 del capítulo 6).

Lo componentes de protocolo optimizado de *CodeSimulink* son los siguientes:

- **`sim_protgen_synchpar`**: protocolo recortado para un componente tipo fuente o sin entradas *CodeSimulink*.

- **`sim_protcnt_synchpar`**: destinado a bloques con N entradas y sólo una salida.

2.4.2. System Generator for DSP

En adelante *System Generator*, esta aplicación diseñada por la empresa *Xilinx, Inc.* presenta también un entorno amigable para el diseño de sistemas

hardware usando *Simulink*. Entre sus características principales encontramos:

- Compatibilidad con el entorno *Matlab–Simulink* que posibilita una co–simulación del sistema.
- El diseño del sistema se lleva a cabo mediante la elección de un conjunto de módulos *Simulink* que son enlazados de forma oportuna. La biblioteca de módulos ampliados comprende desde los componentes más básicos de cada familia de *FPGAs* soportada, hasta módulos más sofisticados que llevan a cabo tareas de procesamiento de más alto nivel (sumadores, filtros, etc...). Con esto, el diseño del sistema hardware–softwareware se lleva a cabo en un nivel muy alto de especificación, accediendo desde éste a la síntesis de dicho sistema de forma automática y sencilla. Es posible crear *scripts*, al igual que en *CodeSimulink*, que interaccionen directamente con los parámetros de cada bloque con objeto de automatizar los procesos de simulación y síntesis.
- Capacidad de síntesis utilizando las siguientes familias de *FPGAs* de *Xilinx*:
  - ⇝ *Virtex–4*
  - ⇝ *Virtex-II Pro*
  - ⇝ *Virtex-II*
  - ⇝ *Virtex-E*
- El bloque ampliado `MicroBlaze` dota al sistema de una interfaz para el diseño de co–procesadores *DSP* (*Digital Signal Processor*) empotrados, compilando el código a un *firmware* que *System Generator* será capaz de co–simular.
- Posibilidad de empotrar módulos *HDL* escritos en *VHDL*, *Verilog* o *EDIF* en un bloque *Simulink* de *System Generator* conocido por `Black Box` o caja negra. Se permite la cosimulación del sistema por medio de del simulador de *HDL ModelSim*.
- Posibilidad de *depuración in–system* o depuración directa sobre la *FPGA* por medio de modulos *System Generator* que *envuelven* los módulos de *ChipScope Pro* (ver página 25).

En la figura 2.4 encontramos un ejemplo de diseño llevado a cabo mediante *Sytem Generator*, en que que se calcula la conversión a escala de grises de una señal tipo *RGB*. Este ejemplo ha sido tomado de la propia *web* del fabricante. Obsérvese que al igual que *CodeSimulink*, es necesaria cierta información adicional relativa a la *FPGA* (selección de patillas, velocidad, etc...) sobre la que se proyectará el sistema y que se integra en un bloque sin entradas ni salidas de nombre *System Generator*, y cuyo único objetivo es albergar estos parámetros.

ChipScope Pro Es un conjunto de bibliotecas y utilidades de *Xilinx, Inc.* que permiten en tiempo real, la verificación lógica *on–chip* de un sistema. Por medio de la inserción en el sistema de un analizador lógico y un analizador de bus, se hace posible el acceso a las señales internas de dicho sistema. También es posible consultar el estado de los procesadores *hard* o *software* incluidos en el chip. El acceso a las señales se opera gracias a la utilidad *ChipScope Pro Logic Analyzer*.

El protocolo de System Generator for DSP.

System Generator al igual que *CodeSimulink* engloba un sistema de especificación de alto nivel que habilita el diseño de sistemas de tipo flujo de datos o *Data Flow*. El empleo de módulos que utilicen sistemas orientados a control o *control oriented systems*, se deja en manos de los bloques *Simulink* puros (los bloques por defecto y sin características hardware) o de los bloques de *System Generator* conocidos por `Black Box` o cajas negras, bloques éstos, como se ha comentado, que definen su funcionalidad por medio de código *VHDL*, *Verilog* o *EDIF* introducido por el usuario.

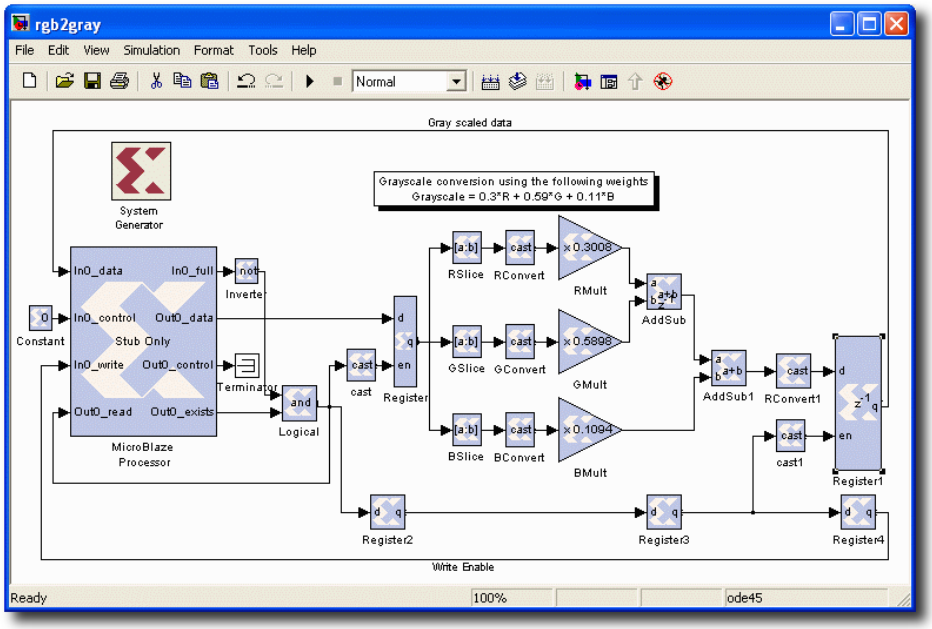

Figura 2.4: Ejemplo de utilización de *CodeSimulink* en el que se efectúa una conversión a escala de grises a partir de una imagen *RGB* proporcionada mediante un procesador empotrado *MicroBlaze*.

2.4.3. Celoxica PixelStreams

PixelStreams es una utilidad y a su vez una biblioteca de primitivas desarrollada por *Celoxica* y destinada al diseño de sistemas de procesamiento de visión (imágenes y vídeo) de forma gráfica, por medio de bloques o cajas (de forma semejante a *CodeSimulink* y *System Generator*). Permite la reutilización y diseño de núcleos *IP* (*IP cores – Intellectual Property cores*) que son tratados como *cajas negras*. El diseño se especifica mediante un editor de esquemas por bloques parecido al que proporciona *Simulink*. El resultado del proceso de síntesis produce un nuevo núcleo *IP* parametrizable destinado a sistemas de visión. Entre las primitivas que pueden encontrarse en *PixelStreams* tenemos:

- Bloques de entrada y salida del flujo de datos.
- Bloques de convolución específicos como el detector de bordes o el detector tipo *sharpening*.
- Filtros no lineales.
- Bloques de control de flujo.
- *Buffers* de imágenes.
- Bloques de conversión entre diferentes espacios de visión.
- Tablas de consulta o *Look-up tables* (*LUTs*) .
- Bloques de aritmética de imágines.
- Fuentes de ruido.
- Bloques de transformación de coordenadas (translación y rotación).

PixelStreams genera un gran *pipeline* o cauce segmentado a partir de la cadena de bloques interconectados. Su modo de procesamiento, al igual que ocurre con *CodeSimulink* y *System Generator* (ver 2.4.1 y 2.4.2), es de tipo flujo de datos o *data flow*. La figura 2.5 muestra una captura de pantalla de esta aplicación en la que podemos observar cómo se diseña un sistema de visión que mezcla dos fuentes distintas de vídeo, a partir del encadenamiento de los distintos tipos de filtros que se proporcionan (clasificados y ordenados a la derecha en la imagen).

Las especificaciones del *API* y las estructuras de datos que maneja están disponibles, con lo que es posible extender la potencia de esta herramienta con nuevos bloques. No obstante, existen algunas limitaciones al respecto de esta herramienta, como es el hecho de que no se pueden definir estructuras de almacenamiento internas en un modelo *PixelStreams*. El procesamiento de *PixelStreams* da como resultado un proyecto en *Handel-C* compuesto del elenco de componentes debidamente enlazados, listo para ser sintetizado por el compilador de *Handel-C* de *Celoxica*. Dado que el entorno de desarrollo *Celoxica DK* puede generar *VHDL* estructural a partir de un proyecto escrito en *Handel-C*, es

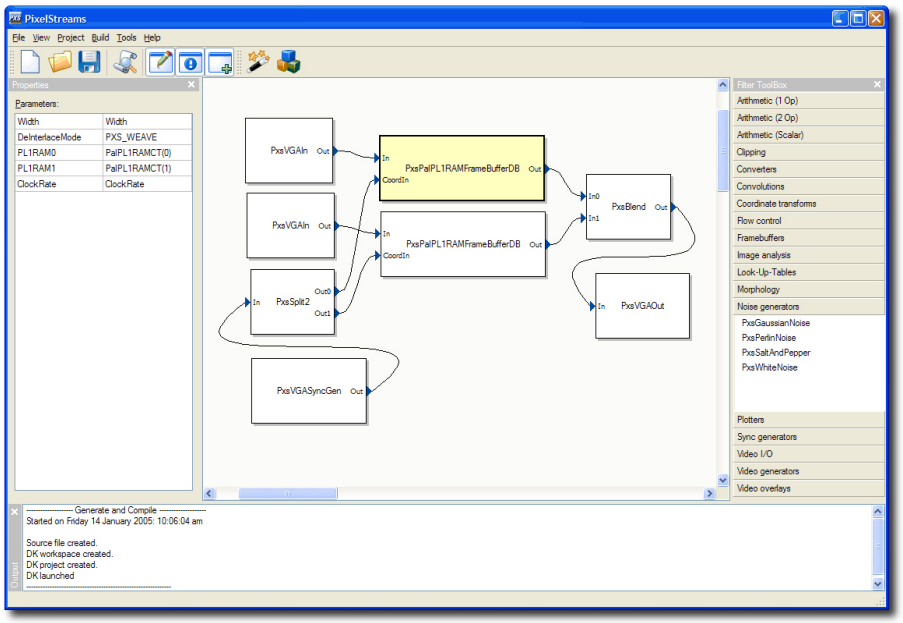

Figura 2.5: Ejemplo de una sesión de edición con *PixelStreams* en el que se diseña un sistema de visión que mezcla dos entradas de vídeo distintas.

posible generar *VHDL* estructural a partir de una descripción por bloques de *PixelStreams*, de forma análoga a los anteriores entornos de co–diseño comentados.

Actualmente la biblioteca de *PixelStreams* soporta las siguientes placas de prototipado *FPGA* de *Celoxica*: *RC10*, *RC200*, *RC203*, *RC250* y *RC300*.

2.5. Conclusiones

En este capítulo se han presentado diversos conceptos, estrategias y herramientas relativas al diseño de sistemas de procesamiento visual, presentándose como un estado del arte en este campo.
Frente a las distintas opciones para llevar a cabo la neuroprótesis que nos ocupa, se ha optado por la realización de dicho trabajo utilizando la tecnología *FPGA* para el prototipado. Para la definición de los modelos de procesamiento visual se ha empleado *VHDL*, el lenguaje más extendido y potente para diseñar sistemas digitales. En relacción a esto, se ha expresado la idoneidad del uso de la herramienta *CodeSimulink* para rentabilizar al máximo las posibilidades del diseño automático de sistemas digitales que se han tenido en cuenta en esta tesis.

Capítulo 3

Situación actual. Modelo de Retina.

El tercer capítulo de la presente memoria realiza un recorrido por la histología y la fisiología de una retina biológica, definiendo el cometido principal del proyecto europeo de investigación en el que se ha desarrollado el trabajo doctoral. La sección 3.1 introduce el capítulo por medio de la definición y el análisis del sentido de la visión. Posteriormente la sección 3.2 realiza un recorrido histológico por las capas de la retina, definiendo las distintas funciones de los componentes más relevantes para el modelado comportamental de la retina. Seguidamente la sección 3.3 comenta los distintos esfuerzos actuales dentro del campo de la neuroingeniería con respecto a distintos tipos de neuroimplantes restitutivos, centrando su atención en los orientados a la visión. Seguidamente la sección 3.4 explora los principales modelos de retina que han constituido la base del modelo final escogido. Por último, la sección 3.5 define los propósitos del proyecto europeo de investigación CORTIVIS*, en el que se ha fundamentado la presente tesis doctoral.*

3.1. Introducción.

El objetivo del presente capítulo es dar una perspectiva de, por una parte, qué es una retina humana y de otra parte, los esfuerzos actuales en imitar el comportamiento de ésta con neuroprótesis destinadas a mejorar la pérdida de visión. El $38\,\%$ de todas las fibras cerebroaferentes de un ser humano pertenecen a nuestro sistema visual. Es por ello que gran parte de todo el procesamiento que realiza nuestro cerebro a lo largo del día, está destinado a extraer información de un mundo decididamente visual como el actual. Una persona cualquiera, tanto en horas de vigilia como en horas de sueño, está inmersa en un mundo con una marcada vocación por la información, que en su mayoría nos llega a través de todo un firmamento de formas (estáticas o dinámicas), colores, de imágenes y en definitiva, todo aquello que pueden captar nuestros

ojos. La inspección visual llevada a cabo por nuestro cerebro es un proceso biológico de extrema importancia hoy en día.

Psicológica y neurológicamente, el ser humano está muy acostumbrado a percibir de una forma muy *sui géneris* a las demás personas y al resto de nuestro universo. Expresiones tan comunes como «*si te entra por los ojos*» dicen mucho del modo como manejamos la información de lo que está «*fuera de nosotros*». Las personas invidentes o con ciertas deficiencias en su sistema visual, terminan en su mayoría por adaptarse a su entorno desde la completa oscuridad o con muy pocos estímulos visuales. Esto da cuenta de la enorme capacidad de adaptación de nuestro órgano más excepcional y maravilloso, nuestro cerebro. Esta capacidad está siendo estudiada detenidamente desde hace pocas décadas con el objetivo de entender y poder buscar soluciones a determinadas deficiencias o traumas neurológicos en el hombre. Actualmente las prótesis cocleares son una realidad que permite oír a personas totalmente sordas dentro de unas determinadas condiciones. La percepción del sonido, así como el escrutinio y entendimiento de éste, es también una tarea del cerebro. Las personas más sordas, suelen adaptarse razonablemente bien a un nuevo mundo de información auditiva, incluso llegan a poder comunicarse mediante el habla después de unos 8–9 meses desde de la intervención quirúrgica. Este hecho proporciona un aliento de esperanza a la hora de intentar reproducir el experimento con el sistema visual.

3.2. El sistema visual de los vertebrados. La retina.

La retina es una delgada capa de células situada en la parte posterior del globo ocular de los vertebrados y los cefalópodos. La función del sistema visual, en el que está integrado la retina, es la de transformar los estímulos luminosos, que llegan en forma de ondas electromagnéticas en el espectro visible, en señales eléctricas compatibles con el cerebro (señales nerviosas). La retina no sólo detecta la luz, sino que como se expone a continuación, juega un papel muy importante en la percepción visual. La imagen de la figura 3.1 muestra la pequeña porción del espectro electromagnético que el ser humano puede procesar con la retina. Esta franja de $400nm$ a $700nm$ de longitudes de onda, configura la lista de los distintos colores que podemos percibir. El sistema visual de los distintos seres vivos, y por tanto también sus retinas, se han adaptado a lo largo de los años de forma eficiente a sus necesidades y características especiales. La tabla 3.1 muestra algunas de la peculiaridades que ha tomado el sistema visual de una serie de animales para adaptarse a su medio [3, 4].

En ella se hace constar que las retinas de diferentes seres vivos se pueden diferenciar entre sí de forma muy notable, no sólo en la cantidad de fotorreceptores que pueden albergar, sino también en todas las propiedades fisiológicas que se exponen en los siguientes sub–apartados.
La luz focaliza en la retina después de atravesar todo el globo ocular entrando por una estrecha apertura llamada pupila, de algunos milímetros de diámetro. La retina debe ser considerada como una parte integrante del sistema nervio-

so, concretamente del cerebro. Desde ella se empieza a procesar la información visual que finalmente percibimos. Está dotada de una serie de conjuntos de células especializadas en percibir el color, el contraste espacial, el movimiento y otras cualidades dependiendo de las especies.

Tabla 3.1: Peculiaridades de los sistema de visión de algunos seres vivos.

Animal	**Cualidad**
Hombre	Rango espectral $400 - 700nm$
Abejas	Rango espectral $300 - 650nm$. Pueden distinguir la luz polarizada.
Buitre	1 millon de fotorreceptores/mm^2. Pueden ver un roedor a $4.5Km$ de altura.
Peces[1]	25 millones de bastones/mm^2. Sólo tienen bastones que están destinados a detectar la bioluminiscencia.
Pingüino	Pueden ver dentro del espectro ultravioleta.
Mariposa *Colias*	Puede discriminar[2]dos puntos separados 30 micras.
Halcón	Puede ver un objecto de $10cm$ a una distancia de $1.5Km$
Calamar	1000 millones de fotorreceptores. Puede distinguir la luz polarizada.
Pulpo	20 millones de fotorreceptores
Gorrión	400.000 fotorreceptores.

Por tanto, la retina no es una mera transductora de la luz en señales eléctricas sino que de hecho, es la que inicia un gigantesco y complejo procesamiento de información visual que se realiza en tiempo–real [3] y con una naturalidad a la que estamos acostumbrados. Hoy en día, con todos los adelantos a que estamos acostumbrados, sigue siendo difícil simular de forma eficiente el comportamiento de la retina con la computadora de sobremesa más potente que podamos comprar. Esto da cuenta del orden de magnitud que hay que afrontar cuando se quiere imitar el comportamiento biológico. Aún cuando un sistema de simulación de la retina por computadora se acerque de forma plausible al comportamiento y velocidad de ésta, surgirán entonces otros problemas para imitar la potencia disipada por la retina e incluso el nivel de ruido acústico[4]. En efecto, los requisitos en nutrientes que un ser vivo necesita para ver son ínfimos comparados con la energía que necesitaría una computadora para llevar a cabo el mismo procesamiento.

[1] De las profundidades abisales.

[2] Con la agudeza visual normal de un ser humano, se puede distinguir dos puntos que se encuentran a 100 micras.

[3] Entiendo tiempo–real el tiempo necesario para que el procesamiento de la información sea tan rápido que estímulo y percepción nos parezcan simultáneos

[4] el procesamiento visual biológico no produce ningún ruido acústico. Vemos *silenciosamente*.

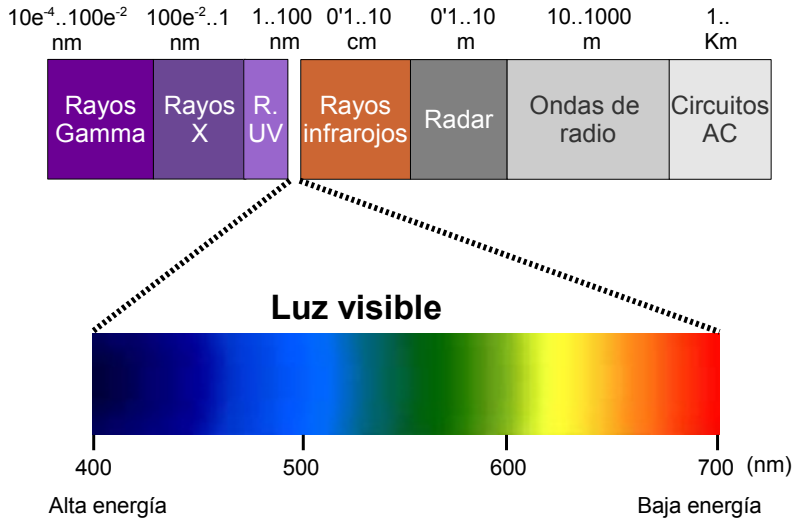

Figura 3.1: Rango de longitudes de onda que puede procesar el ser humano con la vista.

A continuación, el capítulo se va centrar en la retina de los vertebrados, concretamente en la del ser humano o la de los primates.

3.2.1. Histología de la retina.

La retina [5, 6] es una fina capa de tejido nervioso que envuelve una parte del interior del globo ocular que forma un disco circular de unos $42mm$. Hasta llegar a ella (ver figura 3.2), la luz debe atravesar el ojo incidiendo por la córnea y continuando sucesivamente por el *humor acuoso*, el *cristalino*, que actúa como una lente de potencia adaptable, y el *humor vítreo*. Cada una de las regiones anteriores está caracterizada ópticamente por índices de refracción similares, produciéndose el cambio más brusco en el paso del aire a la córnea. Estos índices de refracción configuran un camino óptico que finaliza en la fóvea[5], y no en el eje óptico del ojo (que pasa por la mácula lútea).

La figura 3.3 muestra de forma esquemática las diferentes capas de la retina, con sus células principales y sus funciones fundamentales.

Tiene un espesor que varía desde unos $0.4mm$ en la zona cercana al nervio óptico, hasta los $0.15mm$ de la zona llamada *ora serrata*[6]. Se desarrolla ya desde las primeras etapas de gestación a partir de unas pequeñas bolsitas en la parte frontal del cerebro[7]. Por ésto, la retina debe ser considerada en todo momento como una parte integrante del cerebro. Desde muy temprano, se aprecia una

[5] Donde se experimenta la mayor agudeza visual
[6] Que se sitúa en la periferia de la retina.
[7] vesículas ópticas que nacen directamente del tubo neural

migración de núcleos celulares que va originando una clasificación de la retina por capas de células diferenciadas. Como curiosidad, las células ganglionares, presentadas a continuación, son las primeras en formarse, siendo las últimas los llamados fotorreceptores.

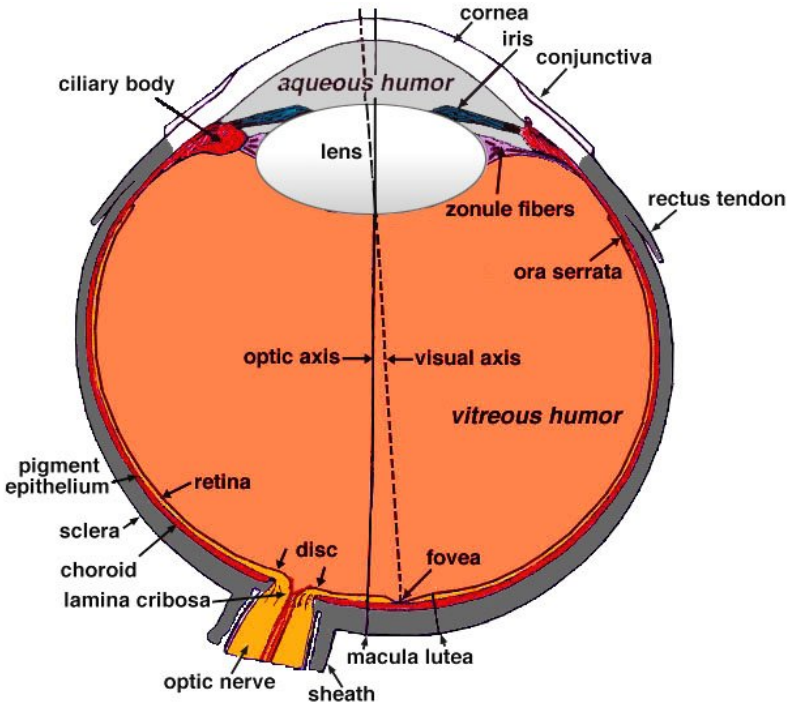

Figura 3.2: Sección de un ojo humano.

Situación y capas principales.

La estructura de la retina se parece a un pastel de tres pisos (ver figura 3.3). Cada capa de la retina está formada por un grupo de cuerpos de células nerviosas ordenados. En cada una de las dos interfaces se realiza la conexión sináptica, es decir el intercambio eléctrico que hace posible el intercambio de información. A estas dos capas se las llama capas plexiformes. Los fotorreceptores, únicas células que se excitan con la luz, se encuentran en la capa más exterior, es decir, la luz tiene que atravesar todas las capas anteriores antes de lograr excitar a los fotorreceptores, tal y como se refleja en la anterior figura. A su vez, la última conexión entre la retina y el nervio óptico tiene lugar en la capa más interna, la capa de células ganglionares.

La figura 3.4 muestra la retina tal y como puede ser observada a través del oftalmoscopio. En ella se pueden distinguir claramente varias zonas con distinta pigmentación (que corresponden a variaciones en la densidad de células

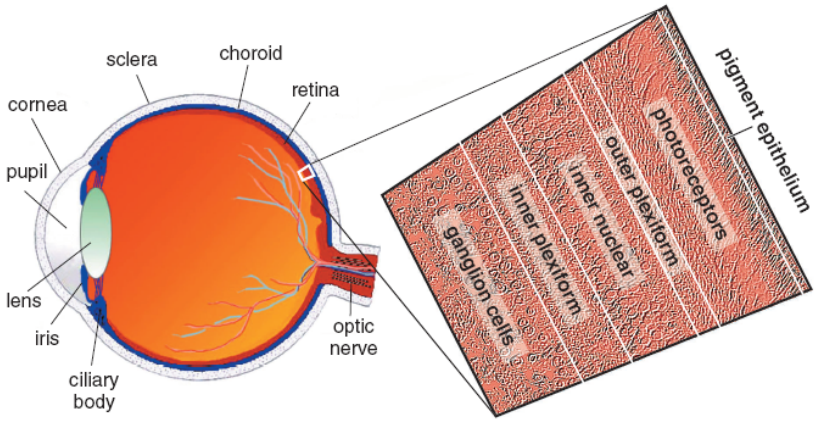

Figura 3.3: Esquema de las capas de la retina.

nerviosas acumuladas) y las arterias principales de la retina. La región más clara pertenece a la zona donde, de forma perpendicular, enlaza el nervio óptico, se la llama «papila» y mide aproximadamente $1.5mm^2$ La mancha más oscura con un pequeño círculo claro es la fóvea que se encuentra a unos 17 grados (4-$4.5mm$) del nervio óptico y hacia la derecha. En la fóvea es donde focalizan los rayos de luz y es allí donde se encuentra la zona de «máxima agudeza visual». Se llama región central de la retina al área de $6mm^2$ que circunda la fóvea. La retina humana tiene en total un área de unos $42mm^2$.

Por su interés en el presente trabajo de tesis doctoral, se define a continuación la agudeza visual:

Agudeza visual Se llama agudeza visual a la capacidad de un ojo para percibir los detalles de una imagen. Está determinada por la función macular, zona de mayor diferenciación de la retina. Una forma de medirla consiste en calcular la distancia a la cual el ojo no puede separar dos puntos próximos de diámetro conocido. Otra forma de estimar la agudeza visual de forma más exacta es mediante el test *Freiburg* (*Freiburg visual acuity test*) [7], que permite realizar la medición mediante una computadora, y cuyo modelo tiene en cuenta varios parámetros psicométricos. La agudeza visual está relacionada de forma muy estrecha con la concentración de células fotorreceptoras en fóvea y la convergencia[8] *fotorreceptor-célula bipolar*. De esta forma menores campos receptivos de células bipolares implican un mayor agudeza visual.

[8]La convergencia determina el campo receptivo de un célula, es decir, el número de células que contribuyen a la entrada de otra dada.

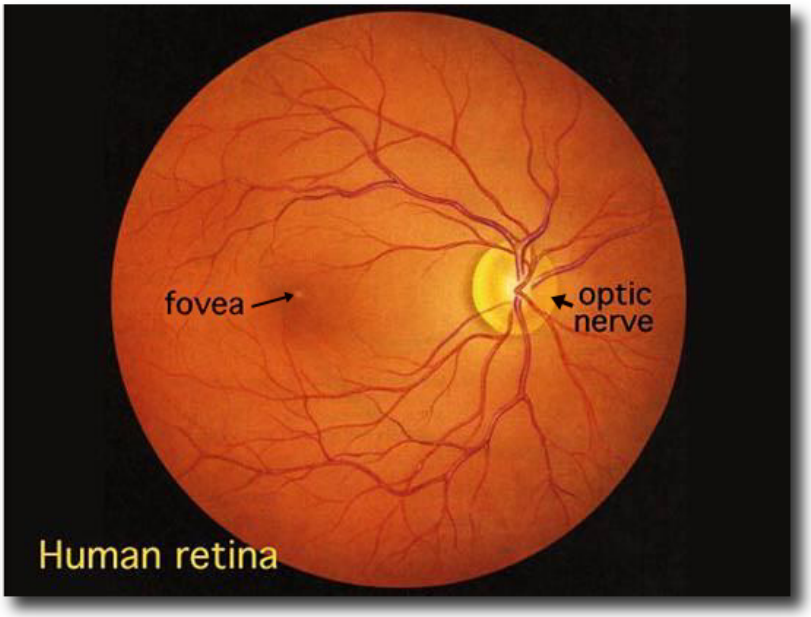

Figura 3.4: Imagen de la retina humana vista a través de un oftalmoscopio.

3.2.2. Células y capas de la retina.

Entender la estructura de la retina es esencial para aproximar su comportamiento. El gráfico 3.5 muestra un esquema por capas de los tipos de células más importantes que se encuentran en la retina humana. Como ya se ha comentado, la retina consta de tres capas de células separadas por dos interfaces o *capas plexiformes*. La capa exterior está formada por células sensibles a la luz, los fotorreceptores. La segunda capa, llamada capa nuclear interna, está integradas por varios tipos de células: $1-4$ tipos de células horizontales, hasta 11 tipos de células bipolares y $22-30$ tipos de amacrinas. Estos números varían en cada especie animal. La tercera y última capa está compuesta por hasta 20 tipos de células ganglionares. Los impulsos producidos por las ganglionares siguen viajando por el cerebro a través de algo más del millón de fibras ópticas que en el ser humano salen de cada ojo. En las dos regiones plexiformes se produce el contacto sináptico o eléctrico entre las dos capas separadas. La primera es la capa plexiforme externa y en ella se producen las sinapsis entre los fotorreceptores y las dentritas de las bipolares y las horizonales. La otra capa plexiforme (la interior), conecta amacrinas y bipolares con la capa de células ganglionares.

Los primeros trabajos y dibujos de estas capas se deben a Santiago Ramón y Cajal [8], quien definió con grado de detalle exquisito la neuroanatomía de la retina. Sus descripciones de la morfología y la interrelacción entre las distintas células de la retina son válidas en la actualidad.

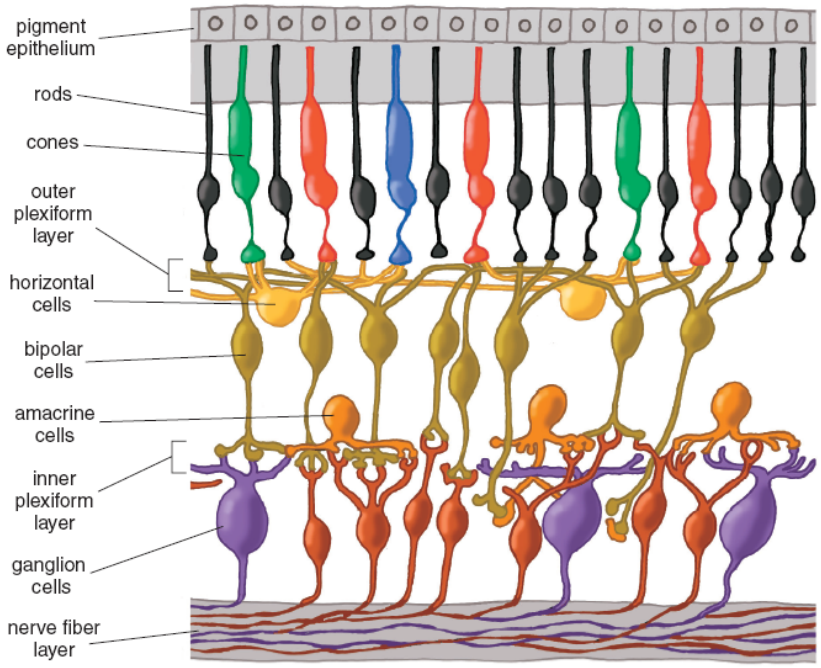

Figura 3.5: Esquema simplificado de interconexión de las células retinianas según un corte perpendicular a la superficie del globo ocular.

En la figura 3.6 podemos observar de forma más realista la disposición por capas de la retina y un esquema de ella en tres dimensiones.

Los fotorreceptores.

Son las células que realizan la transducción de la luz en impulsos eléctricos. Podemos distinguir dos variedades de fotorreceptores: los *conos* y los *bastones*. Todos los mamíferos (en particular el hombre) tienen 2 ó 3 tipos de conos y un único tipo de bastones. Algunos seres vivos no–mamíferos tienen más de tres tipos de conos.

Los bastones sirven para recoger la intensidad de la luz cuando ésta es tenue, durante la noche por ejemplo. Lo conos por otra parte sirven para la visión con luz de más alta intensidad y para distinguir el color. Las adaptaciones en el número y distribución de los fotorreceptores revelan una gran adaptación de los seres vivos a distintas condiciones (como se reflejó en la tabla 3.1). La mayoría de los peces, ranas, tortugas y aves tienen de 3 a 5 tipos de conos. Aunque también hay muchos mamíferos que sólo disponen de 2 tipos de conos (visión dicromática).

La diferencia en el distinto conjunto de tipos de fotorreceptores presentes en especies distintas, es un efecto evolutivo que permite la adaptación a las

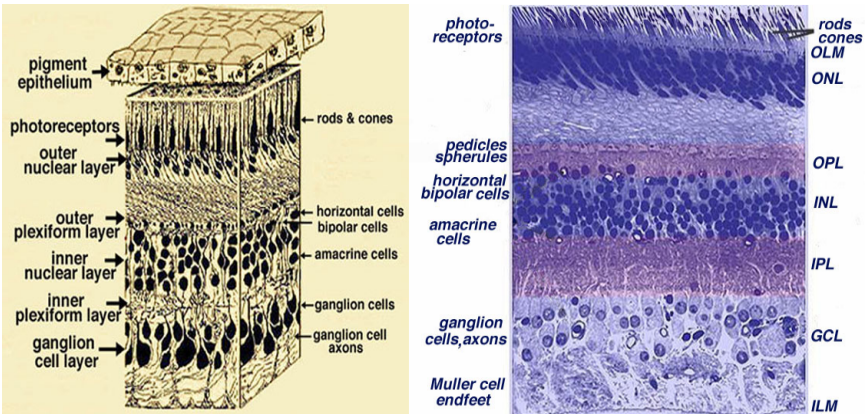

Figura 3.6: Organización de la retina por capas según un corte perpendicular. El gráfico de la izquierda representa un esquema tridimensional de una retina, mientras que a la derecha se muestra una microfotografía real.

características de cáda hábitat concreta.

Con el tiempo la distribución de los fotorreceptores en los mamíferos se ha ido consolidando según un área predominantemente para los conos (área central) destinada a la visión diurna y un área circundante para los bastones. Los primates y las aves rapaces tienen una fóvea muy rica en conos, ellos necesitan mucha agudeza visual y cazan de día. La mayoría de los mamíferos tienen dos tipos de conos sensibles al azul unos y al verde los otros. Los primates y los seres humanos tenemos tres tipos de fotorreceptores sensibles respectivamente a las longitudes de onda correspondientes al rojo, verde y azul. Los conos tienen una estructura cónica, de ahí su nombre. Se encuentran alineados por debajo de una fina capa que los protege. A nivel de la fóvea, donde sólo existen conos, sus cuerpos celulares se sitúan oblicuamente. Los bastones tienen una estructura más alargada y se disponen de forma que rellenan el espacio entre los conos. De forma aproximada podemos decir que los conos tienen un diámetro interno de unas 6 micras frente a las 2 que tienen los bastones. La figura 3.7 muestra algunas imágenes reales de fotorreceptores.

Para entender la organización de los circuitos neuronales en la retina se debe conocer la organización espacial de los distintos tipos de fotorreceptores a lo largo de la retina. La figura 3.8 muestra las distintas densidades para cada una de las células fotorreceptoras. Lo más significativo, como ya se ha comentado, es el hecho de que en la fóvea se situe la máxima concentración de conos (dispuestos de forma hexagonal en esa región). La zona de la papila, que corresponde al *punto ciego*, carece de fotorreceptores.

En la zona de la fóvea, donde más conos tenemos como muestra el gráfico, los conos se disponen de forma oblicua de forma que se pueda aumentar su densidad espacial. Para comprender mejor este hecho, se puede volver a la pasada figura 3.7 y considerar que la parte de la célula que se excita con la luz

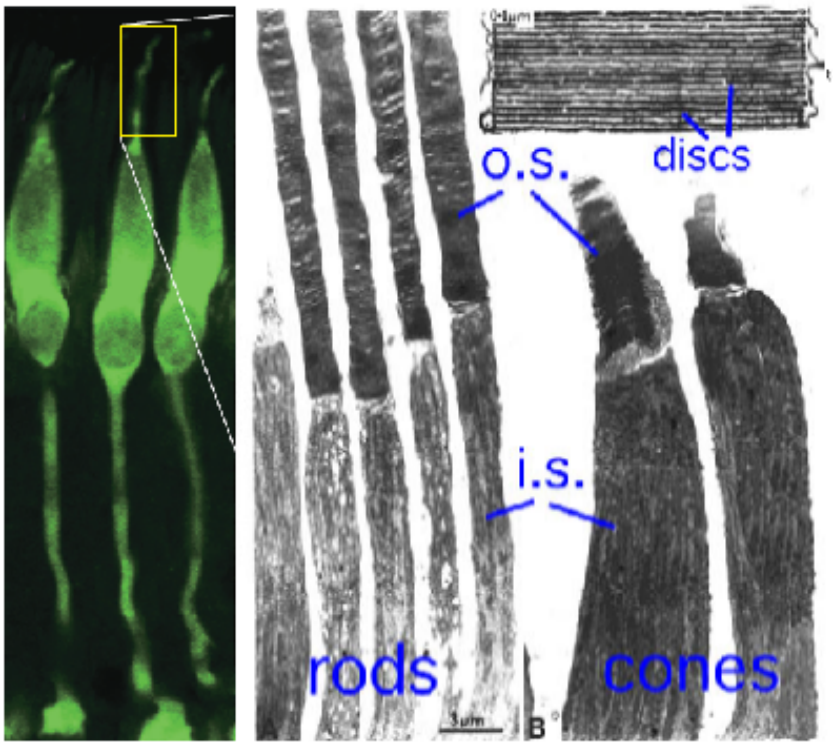

Figura 3.7: Imagen izquierda: fotorreceptores tipo cono de un mono. El apéndice delgado superior es donde se produce la interacción bioquímica entre la luz y la célula. Imagen derecha: Conos y bastones humanos a través del microscopio electrónico. Se puede observar la forma cónica de los conos (a la derecha).

es la *coletilla* (o zona más de menos espesor) que tienen los conos. Un ejemplo de este empaquetamiento puede observarse en la figura 3.7 a la derecha y en la figura 3.9. El ser humano tiene unos 125 millones de fotorreceptores en cada ojo.

Capa plexiforme externa.

Una de las principales característica de la retina, con respecto al procesamiento de la señal luminosa que recibe como entrada, es el nivel de integración al que somete dicha señal. En efecto, la primera integración tiene lugar en la capa plexiforme contigua a los fotorreceptores. Cada capa plexiforme integra 125 millones de señales nerviosas en cada ojo, empezando aquí un sutil y efectivo proceso de compresión de la información y extracción de características, que no podría igualar ni el más potente de los computadores personales actuales. Es aquí donde se realiza la primera *sinapsis* o comunicación eléctrica con las células bipolares y horizontales que veremos a continuación.

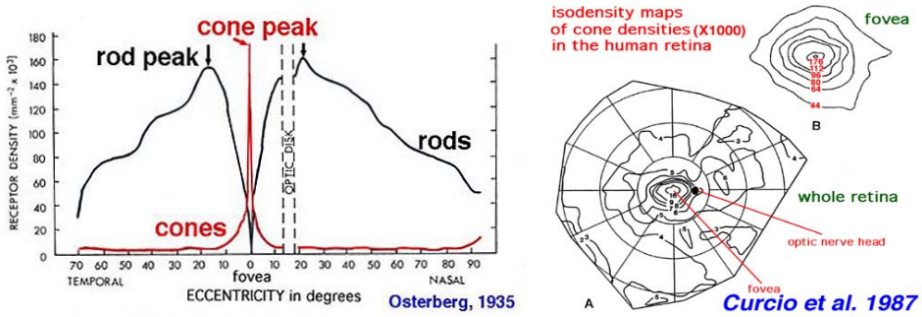

Figura 3.8: Distribución por densidad de las células fotorreceptoras en la retina. A la izquierda la distribución espacial de conos y bastones en un humano. A la derecha un mapa de isodensidad de conos.

En esta capa tienen lugar dos procesamientos de imagen muy importantes:

1. Se separa el flujo de información en dos partes. Una que da cuenta de los objetos que se presentan más brillantes que el fondo circundante y viceversa. Canales *ON* y *OFF* de contraste.

2. Adecuación de forma local a distintos niveles de luminancia.

Células bipolares.

En la retina humana se pueden encontrar hasta 11 tipos diferentes de células bipolares, de las cuales 10 trabajan con nuestros conos y 1 trabaja con los bastones. Estas células conectan los fotorreceptores con las células ganglionares (que se presentarán más adelante). Como el ser humano tiene más bastones que conos en la periferia de la retina, las bipolares asociadas a los bastones son más numerosas. Los diferentes tipos de células se asocian a diferentes tipos de *neurotransmisores* entre otras propiedades. Por regla general en la fóvea cada bipolar integra de $5-7$ conos dentro de su campo receptivo. Esto significa que necesita la contribución de este número de células fotorreceptoras para su activación o disparo. En periferia este *campo receptivo* se amplia a $15-20$ fotorreceptores. Hay varios tipos diferentes de bipolares, siendo los más importantes los que sólo conectan con conos y las que sólo lo hacen con bastones.

En cuanto a su función, se pueden distinguir dos tipos de células bipolares: las despolarizantes y las hiperpolarizantes. Cuando se excitan los fotorreceptores (conos y bastones) unas células bipolares se despolarizan y otras se hiperpolarizan. Esto permite a una parte de las células bipolares transmitir señales *positivas* y a la otra parte señales *negativas*.

Células horizontales.

Son células que definen una interconexión lateral entre las neuronas de la capa plexiforme externa, de forma más concreta, conectan bipolares vecinas

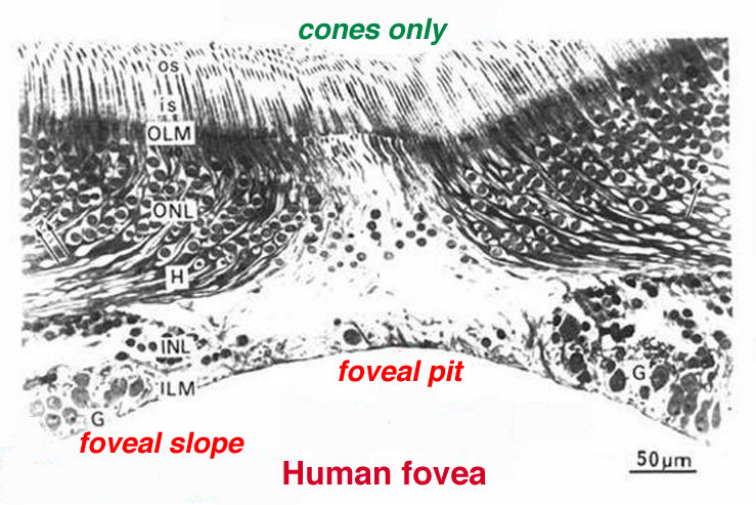

Figura 3.9: Empaquetamiento de los conos en la fóvea. Debido a esta disposición en la zona de la fóvea la retina es más delgada.

entre sí.

En la retina humana existen tres tipos de células horizontales [9, 10]:

- *Tipo I*: No poseen axón y conectan conos rojos y verdes de forma preferencial frente a los azules.

- *Tipo II*: Tienen axón que conectan con los conos azules. Los demás tipos de cono son conectados mediante sus dentritas.

- *Tipo III*: Similares a las del tipo I salvo por su mayor tamaño y el hecho de que evitan la conexión con los conos azules.

La visión en detalle se proporciona mediante estas células, que reciben también la señal de los conos. El campo receptivo suele ser bastante extenso, es decir, se recibe información de varios conos del área circundante. Este campo receptivo es mayor que el de las células bipolares, con lo que las horizontales responden a la luz en un área mayor.

Mientras que las bipolares dan una respuesta de tipo *ON* u *OFF*, las horizontales presentan una respuesta según una señal oponente en la que intervienen los conceptos de centro y periferia de su campo receptivo. Se forma de esta forma una estructura que integra en el «centro» la información *ON* u *OFF* de las bipolares y en la periferia una señal oponente *OFF* u *ON* de las horizontales. De este modo se definen las estructuras *ON* centro – *OFF* periferia y viceversa.

La estructura se puede comprender mejor mirando la figura 3.10. Nótese que el papel de las horizontales es el de influenciar las *entradas* de las células bipolares, que son las que realizan el enlace sináptico con la siguiente capa de

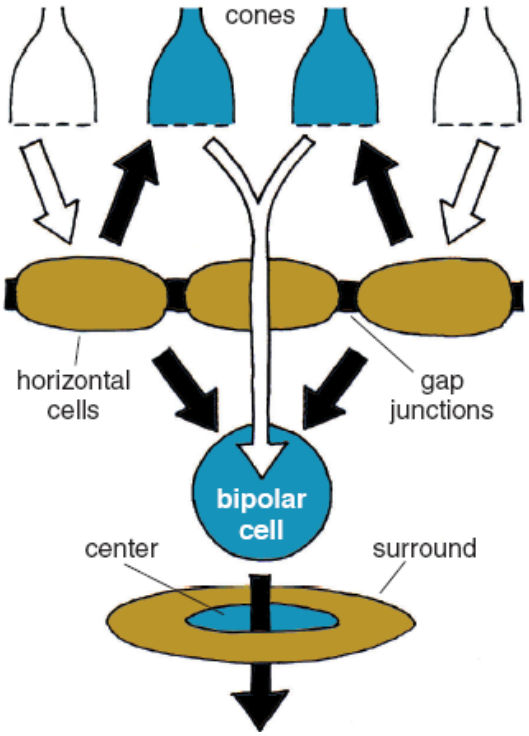

Figura 3.10: Estructura centro–periferia.

células ganglionares. Esta influencia se puede realizar bien directamente actuando sobre la entrada de las bipolares o bien excitando a los conos. En la imagen se puede observar el hecho de que varias horizontales se pueden unir para ampliar su campo receptivo efectivo. El resultado de la actuación de bipolares y horizontales es un filtrado similar al denominado *sombrero mexicano* o *mexican hat* que se comenta más adelante. Este complicado circuito realimentado entre células, así como muchas características del funcionamiento de la retina está siendo objeto todavía de fuerte investigación y debate.

Capa plexiforme interna.

Es la capa que alberga el conjunto de conexiones sinápticas entre ganglionares y bipolares junto con las amacrinas.

En ella se produce la segunda sinápsis de la vía vertical de la retina (consultar la figura 3.5) entre los axones de las células bipolares y las dendritas de las células ganglionares. A este nivel además terminan gran cantidad de prolongaciones de las células amacrinas, que influencían y modulan la información que se transmite a las células ganglionares. Dichas ganglionares constituyen la última etapa del cauce de procesamiento retiniano y envían la información

Canal$_{RB,G}$	o	Canal rojo-verde
Canal$_{RG,B}$	o	Canal amarillo-azul
Canal$_{I,I}$	o	Canal acromático

Tabla 3.2: Conjunto de canales cromáticos de realce espacial que modelan la acción del triplete fotorreceptores-bipolares-horizontales en el ser humano.

codificada hacia el resto del sistema nervioso central a través del nervio óptico.

Filtrado de canales cromáticos oponentes.

Una de las estructuras de procesamiento de la visión más interesantes que se fundamenta en la retina es el filtrado de canales cromáticos oponentes del sistema visual humano. Biológicamente tiene lugar precisamente en la salida de las células ganglionares y se inicia con la combinación de señales procedentes de fotorreceptores de distinta sensibilidad cromática. De esta manera, se puede tener una contribución de excitaciones resultantes de luz roja y azul en el centro del campo receptivo de una ganglionar y verde en la periferia.

En un ser humano, el filtrado de canales cromáticos de la retina puede modelarse mediante tres tipos de contribuciones cromáticas dadas en la tabla 3.2.2:

donde $Canal_{RB,G}$ modela el triplete retiniano fotorreceptores-bipolares-horizontales que realiza un realce espacial rojo-azul en centro frente a verde en periferia, $Canal_{RG,B}$ hace lo propio con el canal amarillo frente al azul y $Canal_I$ lleva a cabo un realce espacial del canal acromático parecido a un detector de bordes. Cada uno de estos canales puede ser de tipo *ON*-centro *OFF*-periferia o *OFF*-centro *ON*-periferia dependiendo si se realza la actividad del centro del campo receptivo frente a la periferia de éste (ver figura 3.10) o se procede de forma inversa.

Como toda célula nerviosa, la respuesta de una célula ganglionar se da en forma de pequeños pulsos de pocos milivoltios siempre y cuando la suma de sus contribuciones supere un cierto umbral. El símil de la carga y descarga del condensador es un buen modelo para describir lo que se observa en el laboratorio.

3.3. Neuroimplantes orientados a visión.

El término neuroimplante o neuroprótesis puede definirse como sigue:

Neuroimplante o neuroprótesis : Ingenio artificial diseñado para interaccionar con el tejido nervioso biológico. Se fundamenta en que la aplicación de ciertos estímulos eléctricos[9] o magnéticos aplicados de forma conveniente al tejido nervioso, pueden llegar a excitar a las neuronas aferentes de forma controlable. Esta excitación nerviosa suele referirse como neuroestimulación.

[9]También referida como electroestimulación

Conviene en este punto, definir el término fosfeno:

Fosfeno Se llama así al artefecto luminoso que se puede percibir mediante una adecuada estimulación neurológica. Esta estimulación se puede producir con estímulos eléctricos (electroestimulación en retina o *cortex* visual) o aplicando una leve presión sobre la retina (presionando por ejemplo el globo ocular).

El éxito de los neuroimplantes cocleares está fundamentado en que suelen restaurar el sentido del oído de forma sorprendente. Este éxito científico–tecnológico ha estimulado la investigación para producir un artilugio capaz de permitir la restauración total o parcial del sentido de la vista a personas con déficits severos de visión. Se pueden encontrar varios tipos de aproximaciones al problema en función del tipo de ceguera que presente el paciente y de la zona que se estimula. Como es bien sabido el sentido de la vista está localizado en una zona o área en el cerebro dedicada exclusivamente a esta tarea. Dicha área toma el nombre de *cortex* visual, situado un poco más arriba de la nuca. Se pueden distinguir también varias zonas: *V1* o *corteza visual primaria* utilizada para visión de forma general, *V2* para visión estéreo, *V3* para medir profundidad y distancia, *V4* para el color, *V5* para percibir el movimiento y *V6* determinar posiciones de objetos de forma relativa. Existen otras muchas regiones que intervienen en el proceso de la visión cognitiva (como por ejemplo los diferentes tipos de regiones de memoria), existiendo una realimentación entre ellas que integra los diferentes procesos cognitivos asociados a la visión. El trabajo de esta tesis doctoral se ha centrado en la zona *V1*, lugar en el que se llevaría a cabo una electroestimulación mediante una matriz de microelectrodos. Una de las características más comunes de los sistema de visión biológicos y en particular de los humanos es que son *retinotópicos*. Dicha característica consiste en que el hecho de que si células fotorreceptoras cercanas son estimuladas, ello producirá actividad en zonas también próximas del *cortex* visual primario *V1*. Esta proyección retinotópica no es exacta (tal como de indica en el apartado 3.5), es decir, las imágenes sufren deformaciones no lineales debido a las irregularidades de la corteza cerebral entre otros motivos. Por ello, si se quiere realizar un sistema visual sintético y proyectar sus resultados sobre la corteza visual, hay que realizar un proceso de *remapping* o adaptación de la imágen, que calcule la *bio–métrica* asociada y proyecte la imagen correcta sobre *V1*. Hablo de *bio–metrica* en el sentido matemático del término, ya que en toda proyección retinotópica subyace una aplicación matemática de $\Re^2 \longrightarrow \Re^2$ fuertemente no lineal y con propiedades análogas a la de una métrica.
Existen varias iniciativas para diseñar neuroimplantes según el punto concreto donde tenga lugar la neuroestimulación:

3.3.1. Implantes en retina.

Son neuroimplantes que tratan de estimular alguna de las capas de la retina mediante un chip al efecto. Se pueden distinguir dos tipos de implantes en retina:

- sobre la retina (***epiretinal***): En este caso la electroestimulación se lleva a cabo con electrodos de superficie, que excitarían en mayor grado a las células ganglionares de la retina.

- debajo de esta (***subretinal***): En los implantes subretinales, los electrodos deben penetrar las distintas capas de la retina hasta llegar excitar la zona de los fotorreceptores.

El procesamiento de la visión en ambos implantes debe ser distinto, ya que se excitan capas de retina que realizan tareas de procesamiento de la imagen diferente. Entre las dimensiones usuales de los electrodos que se utilizan se encuentran matrices de 3×3 y 5×5 electrodos.

Esta cirugía requiere que el nervio óptico se encuentre en buen estado. Las dificultades para operar a este nivel son evidentes, así como los problemas para realizar chips capaces de alimentarse desde el exterior mediante inducción magnética y cuya temperatura de funcionamiento[10] sea biocompatible con la zona de inserción (incrementos alrededor de 0.3 grados centígrados)[11]. Estos implantes no necesitan un *remapping* o redistribución de la imagen (al menos no toma el papel tan importante de los implantes corticales), ya que como excitan directamente la retina, se sirven del aprendizaje previo del individuo antes de perder la visión.
La figura 3.11) muestra un esquema a modo ejemplo de una neuroprótesis visual de epiretiniana[12]. La parte externa de la prótesis se compone de un chip destinado al procesamiento de las imágenes que le son enviadas por medio de una cámara de vídeo. Esta electrónica convierte una escena visual a píxeles. Los datos se envían mediante un sistema de telemetría (basado en láser o en señales moduladas de radio[11]) a la matriz de microelectrodos implantada sobre la retina[12].

También es posible integrar el transductor de luz en señales electricas, la electrónica de procesamiento de la visión, la electrónica de generación del estímulo eléctrico y la matriz de electrodos en el mismo chip. De esta forma se utilizaría la misma óptica del ojo para focalizar sobre el propio implante.

3.3.2. Implantes en nervio óptico.

Se trata de un neuroimplante que conecta directamente con el nervio óptico, siendo un requisito importante, la disfunción total de los centros de procesamiento visual previos: óptica del ojo y retina. De nuevo se requiere que el nervio óptico esté en buenas condiciones para seguir encaminando toda la información visual que, a partir de esta zona, se calcula de forma sintética. Es el proceso quirúrgico más complicado de todos dada la difícil accesibilidad de la

[10] El calor producido a causa del efecto *Joule* hace que la zona de interfaz electrodo–tejido aumente su temperatura, con lo que se corre el riesgo de producir hipertermia.

[11] Como en el caso de la figura, en donde se observa un acoplo inductivo. El globo ocular podría moverse de forma usual al no haber ningún acoplo directo.

[12] Los electrodos que se han empleado en este ejemplo de implante epiretiniano tienen $200\mu m$ de diámetro, están separados unos $400\mu m$ y están configurados en matrices de 3×3 y 5×5 electrodos.

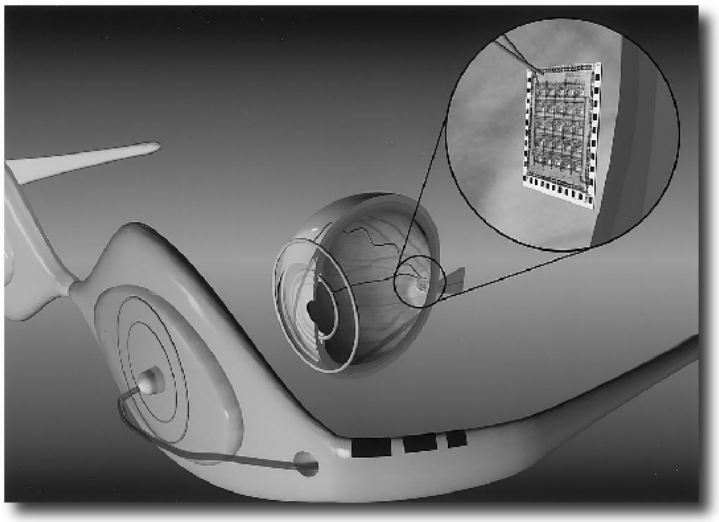

Figura 3.11: Esquema de la realización de una neuroprótesis de retina. La alimentación del chip viene inducida por la electrónica extraocular que contiene las gafas. No hay comunicación directa por cable entre la retina y el exterior, lo que minimiza el riesgo de infección. La electrónica del neuroimplante descodifica la señal y lleva a cabo la electroestimulación de la superficie de la retina.

zona que se quiere excitar. De este modo, aunque es el implante menos estudiado, ofrece unos resultados cuando menos bastante llamativos tal y como se mostrará a continuación.

Este implante conlleva dos inconvenientes importantes[13]:

- La excitación sobre el nervio óptico se realiza por medio de un enrollamiento en espiral que alberga 4 electrodos. Los contactos quedan localizados a 0, 90, 180 y 270 grados de posión angular respectiva. En la zona de excitación de este implante, viaja toda la información del procesamiento retiniano, con lo que se estaría excitando directamente las vías de conducción de un número indeterminado de estructuras que transportan canales de contraste espacial, de procesamiento del color, etc. La morfología del nervio, así como la estructura del electroestimulador, dificulta la especificidad del neuroimplante.

- El tipo y la morfología de los fosfenos producidos son muy dependientes de cada individuo. Como se ha comentado, el fuerte procesamiento de la visión realizado por la retina, integra la información de muchísimas células en canales de procesamiento que viajan por el nervio, la localización de estos canales puede ser a priori impredecible y desordenado, lo que

[13]Como paradigma de este tipo de neuroimplantes, se ha escogido la referencia [13].

conlleva que frente a un mismo estímulo, pacientes distintos no perciban fosfenos similares.

Al contrario que los implantes retinianos o los corticales (que se presentarán más adelante), la excitación de un punto del nervio, no suele producir un punto, sino un fosfeno más complicado[13]. Para ilustrar este efecto, la figura 3.12 muestra la distribución de fosfenos que percibió un paciente implantado siguiendo esta técnica.
El reto es saber *dónde* y *cómo* hay que excitar para poder configurar artificialmente la imagen de un objeto exterior, que como puede verse dado los estímulos que se ven, es una tarea complicada. Para cada paciente habría que definir los fosfenos que puede percibir, y conformar una imagen más complicada a partir de dichos fosfenos base. Las pruebas realizadas hasta la fecha muestran han demostrado que después de un proceso adecuado de aprendizaje, los pacientes implantados pueden distinguir orientaciones de forma plausible.

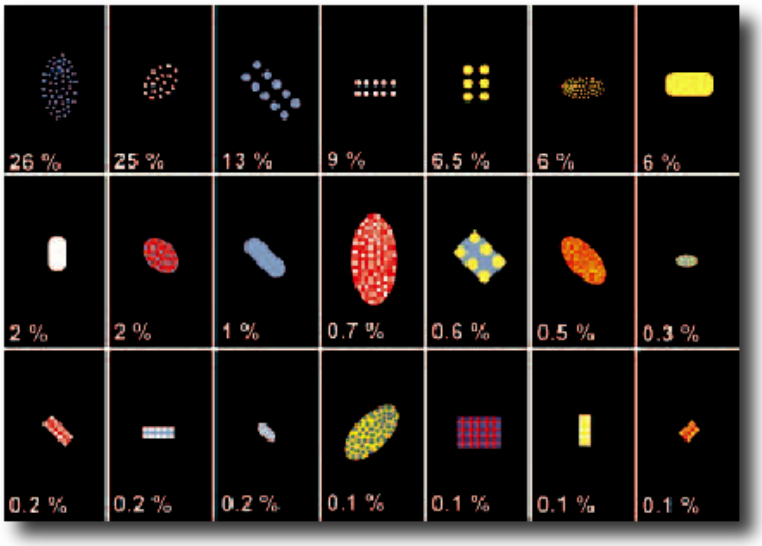

Figura 3.12: Forma de los fosfenos percibidos por un paciente implantado con un neuroestimulador en el nervio óptico. Estos resultados son recientes y han sido realizados en Francia.

En la figura 3.13 podemos ver la radiografía de un paciente implantado según esta técnica. También aquí se alimenta el chip mediante un par de bobinas que generan la energía eléctrica necesaria. Nótese el difícil área de trabajo de esta prótesis.

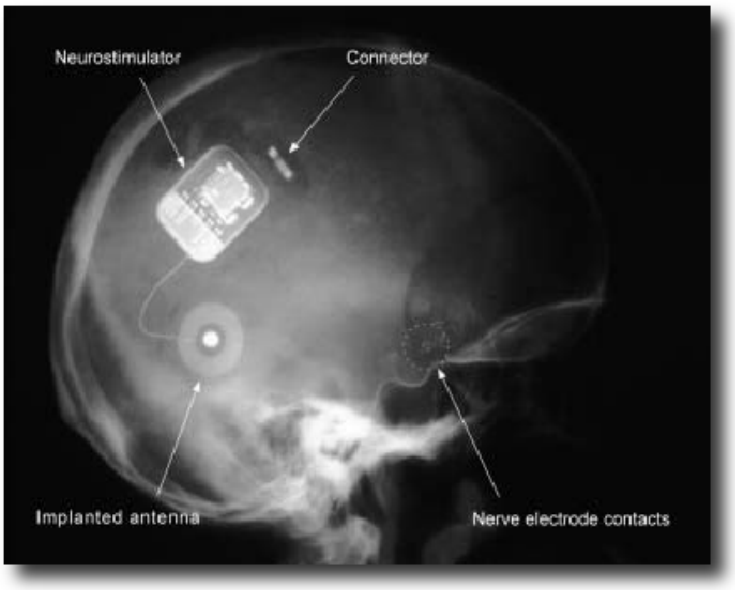

Figura 3.13: Radiografía craneal de un paciente implantado con esta técnica.

3.3.3. Implantes corticales.

En este tipo de neuroimplantes, se plantea una estimulación directa de las áreas visuales del cerebro, con objeto de llegar a producir estímulos visuales. La relacción entre la excitación eléctrica de las neuronas del *cortex* visual con ayuda de electrodos que operen en la zona y la percepción de fosfenos fue demostrada por el doctor *Wm. H. Dobelle* en 1974[14, 15]. Estos implantes no necesitan la contribución de la retina ni del nervio óptico, ya que excitan el área del cerebro destinada directamente a la capacidad de ver (aunque también están involucradas otras áreas). Este *aislamiento ocular* y nervioso, cualifica a los neuroimplantes corticales como una aproximación válida para restaurar la visión en pacientes con daños severos en retina o nervio óptico, por ejemplo una degeneración macular severa o una tumoración en nervio óptico.

Podemos distinguir dos de implantes corticales:

- ***Estimulación superficial***: Se estimulan las neuronas con electronos planos tal y como hace el *Instituto Dobelle*[16]. Este tipo de implante presenta una baja especificidad, ya que los electrodos suelen estimular a una superficie relativamente grande de la corteza visual excitando así a muchas neuronas corticales.

- ***Estimulación directa***: La estimulación se lleva a cabo por medio de matrices de microelectrodos como la diseñada por el profesor Normann[14] de

[14]Matriz de 10×10 microelectrodos de 2 milímetros cuadrados de superfice

la Universidad de *Utah*[17]. Estos implantes presentan mejor convergencia electrodo–nº de neuronas estimuladas (típicamente menos de 5) que los implantes de estimulación superficial. La superficie de estimulación suele ser menor de pocos milímetros cuadrados.

El reto planteado con estos implantes es producir una electroestimulación en corteza visual compatible con el procesamiento de todo el tracto visual anterior (retina, *LGN*, etc.) con objeto de que el paciente pueda percibir escenas visuales capturadas por medio de una o varias cámaras de vídeo. La sección 3.5 amplía la problemática de este tipo de implantes, presentando el neuroimplante cortical objeto de estudio durante el presente trabajo doctoral.

3.4. Modelos de retina

El enorme nivel de paralelismo y granularidad (de grano fino) inherente a todo procesamiento biológico llevado a cabo por el sistema nervioso y en particular por la retina humana, dota a ésta de un rendimiento y una complejidad difícilmente superable por ningún modelo al efecto.

Habida cuenta de esto sin embargo, tradicionalmente se ha recurrido a mimetizar o copiar el funcionamiento y la arquitectura biológica de una retina (y otros complementos del sistema visual) mediante estructuras electrónicas que de por sí tienen una orientación hacia el paralelismo y la granularidad, con un creciente éxito. Nace de esta manera, la *Ingeniería Neuromórfica*, que se define a continuación.

Ingeniería Neuromórfica Término acuñado por *Carver Mead* en torno a 1989 para describir sistemas *VLSI* (*Very Large Scale Integration* analógicos) que contenían circuitos electrónicos analógicos que mimetizaban las arquitecturas neuro–biológicas presentes en el sistema nervioso. Actualmente se ha extendido esta definición para sistemas *VLSI* analógicos, digitales or mixtos que implementan modelos de sistemas neurales y ciertos algoritmos software asociados. Se trata de un área de conocimiento marcadamente interdisciplinar, que está inspirada en la biología, física, matemáticas y la ingeniería electrónica e informática. Las aplicaciones más relevantes se extienden desde procesadores de audio, robots autónomos, sistemas de visión etc. En el trabajo [1] se encuentra esta otra descripción más compacta y general: disciplina que plantea el diseño de sistemas artificiales de computación que heredan ciertas estructuras y propiedades de un sistema nervioso biológico. Así la cooperación entre la Neurociencia y la ingeniería biomédica ha dado lugar a la *ingeniería neuromórica*.

La comunidad científica ha presentado multitud de modelos de retina a lo largo de los años. Estos modelos suelen estar definidos usando el lenguaje y la técnica de la ciencia que propone dicho modelo. En este sentido podemos encontrar modelos puramente matemáticos, modelos electrónicos (definidos mediante circuitos), modelos de bloques interconectados de procesamiento de

señales, etc. En la presente memoria se han tomado dos modelos de retina basados en una descripción circuital (el primero) y en otra matemática (el segundo), habida cuenta de que el modelo final escogido hereda características de los dos, así como de los muchos otros modelos planteados por la comunidad científica. De esta manera, se presentan a continuación los modelos de retina que se han considerado más representativos:

3.4.1. La retina de *Misha Mahowald*.

Una de las pioneras en el campo de la *Ingeniería Neuromórfica* aplicada al diseño de chips de visión es *Misha Mahowald* [18], cuyos trabajos han contribuido a fundamentar y consolidar este área de conocimiento.

La figura 3.14 representa un elemento básico de una retina artificial diseñada por *Misha M.* bajo la dirección de *Carver Mead*. La red de difusión formada por la interconexión de resistencias y el condensador, modelan el campo perceptivo que nutre una célula horizontal, que en este caso toma su entrada de los fotorreceptores (arriba a la izquierda) como una cierta señal de voltaje que depende logarítmicamente de la intensidad recibida. El segundo *triángulo* se comporta como un Amplificador Operacional a Transconductancia (*OTA*), que inyecta una corriente al nodo de la red de difusión que es función de la tensión del foto–transductor. Hasta aquí la estructura modela la capa de células horizontales de la retina. El último *triángulo* se trata de un amplificador diferencial de pequeña ganancia que modela el comportamiento de una célula bipolar, que en el caso de la figura es de tipo *ON*–centro $\wedge$ *OFF*–periferia.

La arquitectura propuesta por *Misha M.* se implementa sobre un chip *VLSI* analógico.

3.4.2. El modelo de Meister – Ammermuler.

Las diferentes capas de la retina humana desarrollan, como se ha mostrado, diferentes tareas de procesamiento de la información visual, que culmina en la capa de células ganglionares que integran toda la información precedente y producen un frente de onda de impulsos o *spikes*[19]. Cada capa tiene diferente respuesta temporal, campos receptivos y sensibilidad cromática. El modelo propuesto por *Meister – Ammermuler* [20, 21, 22, 23] modela la retina mediante un filtrado espacio–temporal de la entrada, con un control local de ganancia del contraste.

La arquitectura propuesta está compuesta de una entrada $I(\mathbf{x}, t)$ que consiste en un estímulo espacio–temporal de patrones de actividad que modelan la excitación de los fotorreceptores. El modelo consiste en un conjunto de filtros *paso–banda* espaciales ($S(\mathbf{x})$) y un filtro *paso–alta* temporal ($T(t)$). La actuación del filtro espacio–temporal se supone factorizable en una parte espacial y otra temporal tal como indica la ecuación 3.1.

La respuesta espacial $S(\mathbf{x})$ se define mediante un filtrado del tipo *diferencia de gaussianas* o *DoG* y modela el procesamiento de la triada fotorreceptores-bipolares-horizontales. Concretamente el comportamiento *ON*–centro $\wedge$ *OFF*–

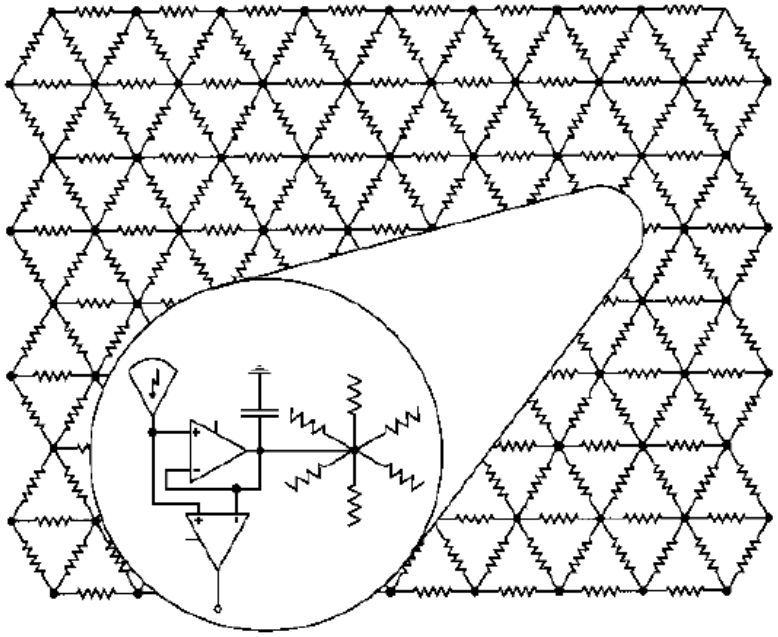

Figura 3.14: Modelo de Retina propuesto por *Misha Mahowald* y *Carver Mead*.

periferia (y viceversa) de las células horizontales se modela correctamente con parámetros adecuados del filtro *DoG* que lo transforman en un *filtro sombrero mexicano* (*Mexican hat filter*) como el que se muestra en la figura 3.17.

$$Retina(\mathbf{x}, t) = S(\mathbf{x}) \cdot T(t) \tag{3.1}$$

3.4.3. El modelo de visión de Itti.

El modelo computacional de Laurent Itti [24] de atención selectiva se basa en aplicar una determinada combinación de filtros espacio-temporales sobre una imagen dada, procesarlos mediante ciertas reglas de aprendizaje que utilizan ciertos caminos de realimentación de los mapas de saliencia y determinar así los lugares más destacados en cuanto al mecanismo biológico de atención.

En el banco de filtros que se utilizan, se encuentran filtros multicromáticos y de intensidad (*ON* y *OFF*), filtros sensibles a la orientación, filtros de cálculo de la disparidad en estéreo, detectores de sombra, de movimiento etc. El modelo establece un proceso competitivo entre ellos para generar una combinación de características, cuyo mapa de saliencia establece un orden de prioridad entre los lugares más distinguidos o llamativos de la imagen.

Laurent Itti descubrió entre otras cosas, que el mapa de saliencia dependía mucho del contexto de la imagen, que la inhibición del lugar ganador era una

estratégia crucial para determinar el orden entre los lugares de atención, y que el *movimiento de los ojos* jugaba también un papel importante.

3.5. El proyecto europeo *CORTIVIS*

El consorcio *CORTIVIS* tiene como principal objetivo desarrollar una neuroprótesis visual que actué a nivel cortical, destinada principalmente a personas que hayan sufrido una pérdida total de la visión. El problema planteado es complejo y en la actualidad, tal y como se ha mostrado en la sección anterior, existe un gran número de laboratorios en todo el mundo trabajando en direcciones similares, evaluando prótesis que actúan a nivel de los siguientes centros del tracto visual: retina, nervio óptico y cortex visual primario.
A diferencia de las prótesis cocleares, que han alcanzado un elevado grado de éxito, siendo actualmente una opción quirúrgica muy válida para la rehabilitación de muchos casos de sordera, las prótesis visuales tienen ciertas dificultades añadidas que han provocado un retraso en su desarrollo. Estas dificultades se pueden catalogar según los casos siguientes:

- *De tipo técnico*: motivadas fundamentalmente por el elevado número de electrodos de estimulación necesarios para producir una percepción visual con una resolución útil.

- *De carácter quirúrgico*: Dado que dichas prótesis deben implantarse en centros más próximos al cerebro y por tanto más inaccesibles y con mayores riesgos relacionados con la bio-incompatibilidad.

En definitiva, el diseño de una neuroprótesis cortical es un reto difícil de superar, pero en el que pequeños avances suponen una gran esperanza para las personas (cada vez más numerosas, debido a accidentes y al envejecimiento progresivo de la población) que pierdan la visión, órgano sensorial que más información aporta en la vida cotidiana. No obstante, deberá transcurrir algún tiempo, puede que en torno a 5–10 años, para alcanzar en este campo una madurez similar a la que disfrutan actualmente los implantes cocleares.
Entre de las tareas inicialmente planteadas en *CORTIVIS* se encuentra la implementación de un sistema de procesamiento bioinspirado de información visual, que funcione en tiempo–real y que sea capaz de producir estímulos nerviosos similares a los recibidos en el cortex visual primario de una persona vidente normal. En el contexto de procesamiento de la visión se entenderá por tiempo–real lo siguiente[15]:

[15]Realmente el concepto de tiempo–real viene ligado a la cualidad de un sistema de procesamiento, de ofrecer resultados válidos a intervalos de tiempo conocidos. Con esto, dicha cualidad no tiene nada que ver con la velocidad de procesamiento de un sistema concreto. No obstante, es usual que en el ámbito de las tareas de modelado hardware de algoritmos de procesamiento de la visión, cuyo propósito es el de acelerar la velocidad de procesamiento de éstos, se confunda deliberadamente tiempo–real con velocidad de procesamiento. Éste, como se indica, es también el sentido que se le ha dado en la presente memoria.

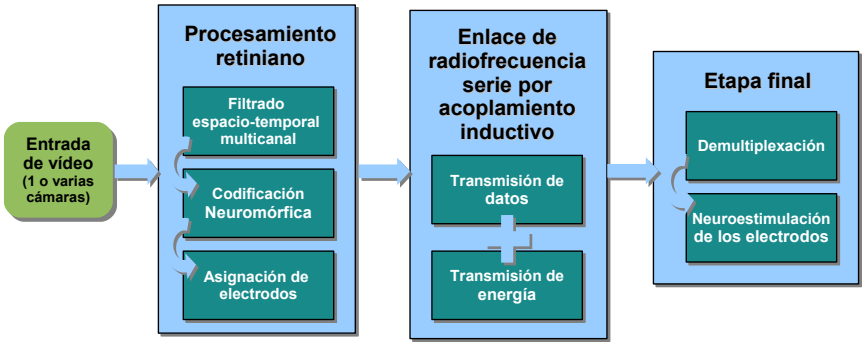

Figura 3.15: Esquema general del sistema de procesamiento de retina y transmisión de información visual para neuroestimulación del proyecto europeo *CORTIVIS*.

tiempo–real Velocidad de procesamiento tal que pueda finalmente llegar a proporcionar una percepción visual casi continua.

El prototipo inicial de retina artificial, será empleado para la experimentación y desarrollo del resto de la electrónica asociada con la prótesis, y finalmente debe poderse integrar en un chip específico, particularizable para cada paciente. Por este motivo es esencial generar descripciones hardware reutilizables, que inicialmente se proyecten en una lógica reconfigurable, y dado el caso puedan fácilmente sintetizarse para una tecnología de *ASIC*.
En la organización del diseño del sistema protésico completo llevada a cabo por el proyecto *CORTIVIS* (ver figura 3.15) se pueden distinguir los siguientes bloques:

- *Bloque de procesamiento retiniano*:

  En el que la información visual de entrada es sometida a un procesamiento bioinsbirado que modela el comportamiento de la retina. En dicho proceso se distinguen a su vez las siguientes etapas:

  ⇝ *Filtrado espacio–temporal multicanal*: que corresponde al filtrado paso–banda espacio–temporal que efectúa la retina con una adecuada combinación de los canales cromáticos rojo (R), verde (G) y azul (B).
  Por tanto, inspirado en los modelos de retina anteriores, se ha considerado en el seno de *CORTIVIS*, que la acción del filtrado multicromático espacio–temporal de la retina puede describirse matemáticamente mediante una combinación lineal de un conjunto *bioinspirado* de funciones, entre las que se destacan las siguientes[16]:

  - *Función gaussiana* (ver A.1).
  - *Función DoG* o diferencia de gaussianas (ver A.1.3).

[16] Ver Apéndice A

- *Función LoG* o laplaciano de gaussianas (ver A.3).

Por tanto, la descripción del sistema general de procesamiento retiniano toma la forma de la ecuación 3.6, definida mediante el siguiente conjunto de expresiones matemáticas:

$$\begin{aligned} f_1 &= DoG(\sigma_{11}, \sigma_{12}, L_1, I_{11}, I_{12}; \ldots) \\ f_2 &= DoG(\sigma_{21}, \sigma_{22}, L_2, I_{21}, G_{22}; \ldots) \\ &\vdots \\ f_N &= DoG(\sigma_{N1}, \sigma_{N2}, L_N, I_{N1}, I_{N2}; \ldots) \end{aligned} \tag{3.2}$$

$$\begin{aligned} f_{N+1} &= Gauss(\sigma_{N+1}, L_{N+1}, I_{N+1\,1}; \ldots) \\ f_{N+2} &= Gauss(\sigma_{N+2}, L_{N+2}, I_{N+2\,2}; \ldots) \\ &\vdots \\ f_{N+M} &= Gauss(\sigma_{N+M}, L_{N+M}, I_{N+M\,M}; \ldots) \end{aligned} \tag{3.3}$$

$$\begin{aligned} f_{N+M+1} &= LoG(\sigma_{N+M+1}, L_{N+M+1}, I_{N+M+1}; \ldots) \\ f_{N+M+2} &= LoG(\sigma_{N+M+2}, L_{N+M+2}, I_{N+M+2}; \ldots) \\ &\vdots \\ f_{N+M+L} &= LoG(\sigma_{N+M+L}, L_{N+M+L}, I_{N+M+L}; \ldots) \end{aligned} \tag{3.4}$$

$$\begin{aligned} f_{N+M+L+1} &= f_{conv}(\sigma_{N+M+L+1}, L_{N+M+L+1}, I_{N+M+L+1}; \ldots) \\ f_{N+M+L+2} &= f_{conv}(\sigma_{N+M+L+2}, L_{N+M+L+2}, I_{N+M+L+2}; \ldots) \\ &\vdots \\ f_{N+M+L+P} &= f_{conv}(\sigma_{N+M+L+P}, L_{N+M+L+P}, I_{N+M+L+P}; \ldots) \end{aligned} \tag{3.5}$$

$$Filtrado_{Retiniano} = a \times f_1 + b \times f_2 + c \times f_3 + \ldots \tag{3.6}$$

donde σ_i es la desviación típica de la i-ésima gaussiana, L_i es la dimensión del kernel de convolution[17] de la i-ésima función e I_{ij} corresponde con la j-ésima entrada de la función i-ésima. Nótese que no se pierde generalidad al usar máscaras cuadradas, porque siempre se podrá extender una máscara rectangular de dimensiones $L \times M$ donde $L \neq M$ a una máscara cuadrada de dimensiones $N \times N$ donde $N = max(L, N)$.

Como se observa, se ha extendido el rango de funciones biológicas a todas las obtenidas mediante el operador convolución (funciones $f_{N+M+L+1}$ $\ldots f_{N+M+L+P}$), ya que es precisamente la aplicación de este operador, la

[17] De esta forma $L_i = 7$ definiría una máscara de convolución cuadrada de tamaño 7×7.

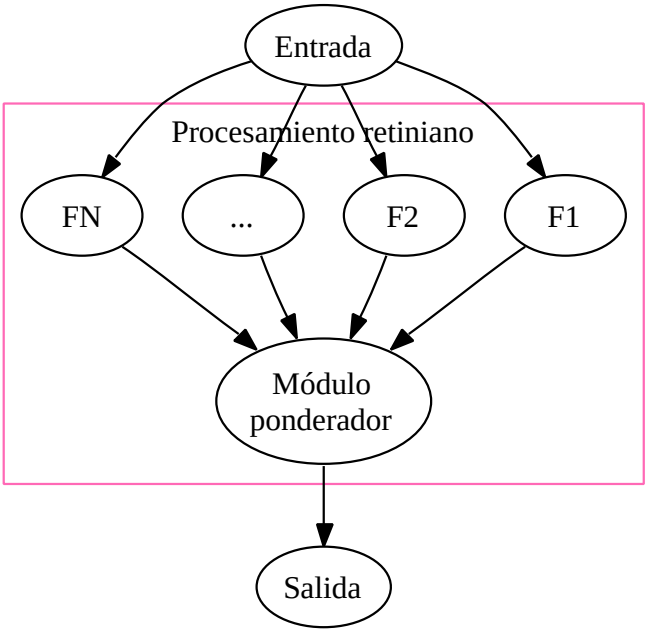

Figura 3.16: Esquema de procesamiento retiniano en donde se manifiesta que todos los filtros se calculan de forma paralela para atender a los requisitos de tiempo–real que se manejan.

que mejor modela la red neural de conexiones sinápticas. De esta manera, podríamos extender el modelado retiniano para, por ejemplo incluir otros aspectos del procesamiento del *cortex visual* tales como los filtros direccionales de *Gabor* que producen una respuesta selectiva según una orientación controlada.

En el diseño del procesamiento retiniano, la aplicación de todos los filtros se realiza en paralelo dado los requisitos de cómputo en *tiempo–real*. La figura 3.16 esquematiza la actuación de este módulo.

Con objeto de ilustrar la plausibilidad biológica de la función *DoG*, la función más importante a la hora de modelar matemáticamente la retina, la figura 3.17 muestra un ejemplo de la actución de dicha función para mimetizar el comportamiento *ON*–centro *OFF*–periferia de la triada *Fotorreceptores–Bipolares–Horizontales*. La gaussiana de la izquierda se aplica sobre una combinación lineal dada de los canales cromáticos, por ejemplo sobre 2 veces el canal rojo ($2 \cdot R$) y la derecha podría aplicarse al canal azul (B). El resultado es el filtrado multicromático esquematizado en la función de abajo, cuya forma de sombrero mexicano amplifica las zonas de la matriz de entrada donde hay más rojo en el centro (contribución positiva) en comparación con el verde de la periferia (contribución

negativa).

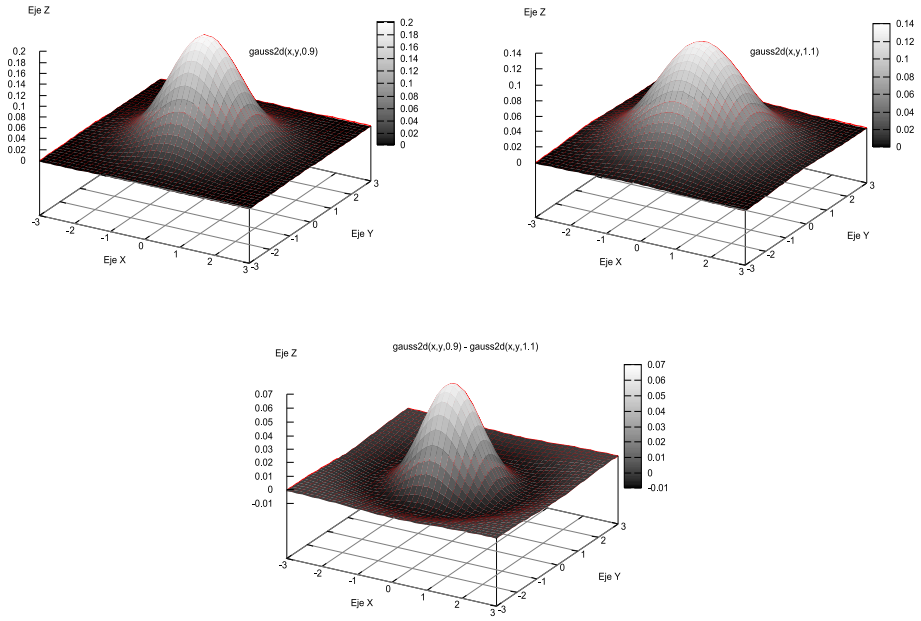

Figura 3.17: Esquema de la actuación del filtrado *DoG* multicanal. Las gaussianas de arriba se aplican sobre sendas combinaciones lineales de los canales cromáticos. El aspecto del filtro resultande de la sustración de las dos gaussianas se muestra en la imagen de abajo, que modela la triada *Fotorreceptores–Bipolares–Horizontales*

⇝ *Codificación neuromórfica*: Procesamiento que corresponde a la actuación sobre el filtrado anterior, de un modelo neural del tipo *integra y dispara integrate and fire*.

El modelo empleado para la codificación neuromórfica es una adaptación del modelo original de *integración y disparo* que se describe con más detalle en [25].

El comportamiento del modelo se define mediante tres parámetros:

- El umbral de disparo o *threshold*.
- El potencial de relajación o *resting potential*.
- El término de pérdidas o *leakage term*, que tiende a anular la respuesta en ausencia de estímulos en la entrada.

La figura 3.18 muestra un diagrama de bloques que describe el funcionamiento del modelo de *integración y disparo*, también se ilustra la operación básica de este módulo utilizando un estímulo sencillo. El módulo de disparo produce una secuencia de impulsos (*spike train*) que se generan

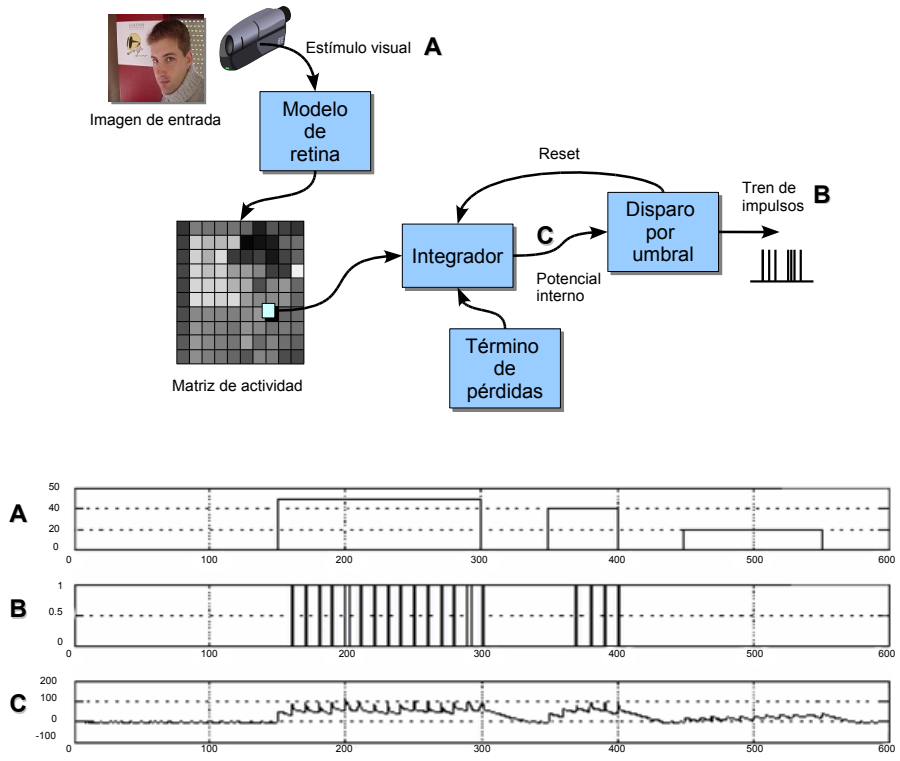

Figura 3.18: Esquema de la implementación del modelo neuronal tipo *Integra y dispara* que se aplica a cada componente de la matriz de actividad resultante de aplicar un determinado modelo de retina a la imagen que se tiene. La gráfica *A* representa el diagrama temporal de los valores de la matriz de actividad

cuando el *potencial interno* supera un cierto valor umbral. El módulo de integración se encarga de acumular la actividad que le llega, incrementando el *potencial interno*, mientras que el módulo *término de pérdidas*, define un cierto decremento de potencial, que en este caso es independiente de la entrada al integrador.

La parte inferior de la imagen muestra el resultado de la simulación obtenida mediante la aplicación del estímulo mostrado en la primera señal temporal. El segundo diagrama temporal muestra el registro de la actividad de salida de impulsos. Puede observarse el efecto de que el tiempo del primer disparo es inversamente proporcional a la intensidad del estímulo de entrada y que si la intensidad es inferior a cierto valor, no se produce el disparo.

La figura 3.19 muestra una implementación mediante bloques de *System Generator* de un modelo neuronal de *integración y disparo* descrito más de-

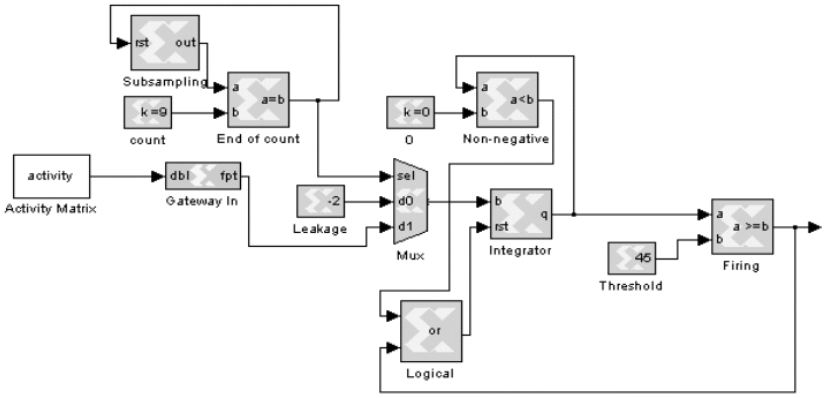

Figura 3.19: Diagrama por bloques para modelar el disparo de una neurona (*spiking neuron*) descrito mediante módulos de *System Generator*

talladamente en [26]. En este caso el bloque `Activity Matriz` describe el disparo de una neurona que toma su entrada de un elemento de la matriz de actividad.

⇝ *Asignación de electrodos*: Etapa que configura la asignación o *mapping* de la matriz de información neuromórfica, con objeto de adaptarse a las características propias de la corteza visual de cada indivíduo. Para entender correctamente este módulo conviene introducir el término *proyección retinotópica*:

- *Proyección retinotópica*:

  Es un hecho comprobado que si se excitan dos fotorreceptores próximos entre sí, el procesamiento del sistema visual biológico proyectará ese estímulo en zonas también próximas de la corteza visual. En este sentido, la proyección de la imagen desde la retina al *cortex* visual primario se dice que es *retinotópica*.

  De esta forma el tracto visual biológico transforma planos en planos. No obstante la relación no es lineal, ya que intervienen muchos factores fisiológicos en la actuación del sistema de visión, como por ejemplo la forma del *cortex* visual, que no es plana.

Las deformaciones que conlleva la proyección retinotópica, dependientes del paciente y de la zona concreta del posible implante, pueden ser corregidas artificialmente mediante una asignación oportuna de las direcciones de destino de los pulsos procedentes de la retina. Este es pues, la misión de este módulo.

- *Enlace de radiofrecuencia serie por acoplamiento inductivo* En esta etapa la información fruto del procesamiento retiniano se translada a la electrónica

de estimulación de los microelectrodos, situada encima de la corteza visual del individuo por medio de un sistema de telemetría. Para su translado se utiliza la asepsia de un acoplamiento inductivo con el propósito de que no exista ninguna vía transcutánea abierta de contacto con el exterior del cerebro, evitándose así múltiples problemas derivados de posibles infecciones. Este enlace sirve tanto para transladar la energía necesaria para el correcto funcionamiento de la electrónica sobre la corteza visual, como para trasmitir el procesamiento retiniano mediante un protocolo serie basado en *Representación de eventos mediante direcciones* o*AER* (*Address Event Representation*). Mediante el algoritmo *AER* es posible estimular de forma neurológicamente *simultánea* varios electrodos, definiendo estímulos simultáneos en este contexto como:

Estímulos simultáneos : Conjunto de estímulos comprendidos en una ventana temporal de duración inferior a pocos milisegundos.

La matriz de microelectrodos escogida para los primeros prototipos, es la diseñada en la Universidad de *Utah* [17], que emplea microelectrodos de silicio con una separación de unos $400\mu m$, distancia del mismo orden de magnitud que la separación entre las neuronas de la corteza visual *V1*.

- *Etapa final* Etapa final de conducción y adecuación biológica del resultado del procesamiento retiniano. En esta última etapa, para cada evento o *spike* se genera una forma de onda de estimulación eléctrica que se dirige adecuadamente hacia el microelectrodo oportuno.

Nuestro trabajo en *CORTIVIS* se ha centrado en diseñar un bloque de codificación de la salida retiniana en forma de impulsos que puedan producirse con ciertas cadencias y con desfases temporales concretos, y cuya proyección sobre la matriz de microelectrodos de estimulación sea totalmente reconfigurable (de aquí el empleo de tecnologías *FPGA*).

3.6. Conclusiones

En este capítulo se ha introducido la fisiología y la histología de la retina, resumiendo los aspectos más importantes para entender el procesamiento de la informacion visual retiniana. Posteriormente se ha presentado el proyecto europeo de investigación *CORTIVIS*, proyecto en el que el doctorando ha participado de forma activa y en el que se ha fundamentado el trabajo de la presente tesis doctoral.

El reto pues del proyecto *CORTIVIS* es mimetizar una retina biológica, parte integrante del cerebro en donde el concurso de varias capas de neuronas, produce un realce de contrastes espacio-temporal de la información visual, y cuya salida es notablemente independiente de las condiciones de iluminación.

El flujo de imágenes se toma a partir de una o varias cámaras digitales, y es procesado por un sistema hardware. El proceso de extracción de información visual adecuada se realiza sin el concurso de ningún recurso biológico.

El modelo de procesamiento retiniano que se ha adoptado comparte características de múltiples modelos referenciados en la bibliografía, adaptados para el tipo de estimulación que se persigue (neuronas biológicas, resolución restringuida debida a limitado número de microelectrodos que se pueden emplear, etc.) y con la restricciones propias del modelo de comunicación a utilizar.

Dado el alto grado de paralelismo inherente al problema, y la necesidad de obtener un rendimiento y velocidad adecuados para biocompatibilizar el proceso, se ha optado por emplear soluciones basadas en *FPGA*.

Capítulo 4

Plataforma de diseño de sistemas de visión

El presente capítulo fundamenta, presenta y articula los tres capítulos siguientes (5, 6, 7), integrándolos en una propuesta de plataforma informática *para el diseño de sistemas digitales de visión y otros más generales. La sección 4.1 define el término* plataforma informática *tal y como es utilizado a lo largo de toda la memoria. En la sección 4.2 se presenta la plataforma informática de especificación, test, validación y síntesis digital de sistemas de cómputo orientados al flujo de datos, articulándola en tres módulos fundamentales: simulación, modelado de hardware y síntesis automática. También muestra los lenguajes de programación utilizados para desarrollar dicha plataforma, así como idoneidad de cada uno de ellos según el caso. La sección 4.3 describe y fundamenta la elección del proyecto* Mono *como plataforma de desarrollo y ejecución del módulo de síntesis automática. Por último la sección 4.4 concluye el capítulo realizando un breve resumen del mismo.*

4.1. Introducción

El término *plataforma informática* suele tener distintas acepciones dependiendo de la disciplina que la utilice. Además en ingeniería informática suele usarse con más de un significado, según el campo curricular de que se trate en cada caso (Arquitectura y Tecnología de Computadores, Lenguajes y Sistemas Informáticos, Ciencias de la Computación e Inteligencia Artificial, etc...). Por esto, y para unificar criterios, se presenta a continuación la definición de *plataforma* que ha adoptado la presente memoria de tesis doctoral:

Plataforma informática: En el entorno de la ingeniería electrónica e informática, se puede definir *plataforma* como la unión de un determinado hardware, su software asociado y el conjunto de las metodologías de programación y modelos de utilidad que los integran. Como ejemplos de plataformas se tienen la *plataforma x86 + el S.O. Linux o Windows.* y la *plataforma*

ARM + Mono, .Net o Java.

Este capítulo presenta y fundamenta una propuesta de plataforma de especificación, test, validación y síntesis digital de sistemas de cómputo orientados al flujo de datos y particularizada de forma oportuna para el diseño de sistemas de visión.

En adelante se utilizarán los siguientes términos relacionados con la palabra *plataforma* con los significados siguientes:

Plataforma software : Para referir la parte *software* de una plataforma informática concreta o para englobar un conjunto extenso de bibliotecas y programas orientados hacia un objetivo común.

Plataforma hardware : Para distinguir la parte *hardware* de una plataforma informática. Con este sentido se entenderá que tanto una tarjeta *PCI* con una o varias *FPGAs* integradas o la arquitectura *x86* son ejemplos de plataformas hardware.

Plataforma : Para referir el concepto de plataforma informática definida previamente.

4.2. Propuesta de plataforma para el diseño de sistemas de visión

4.2.1. Esquema general

La plataforma que se propone está integrada por tres módulos diferenciados que se describen con detalle en los capítulos 5, 6 y 7. Los módulos de dicha plataforma son los siguientes:

- *Módulo de simulación funcional*: Descrito en el capítulo 5, hace el análisis de una herramienta ad–hoc de nombre *Retiner* [27, 28, 26] cuyo principal objetivo, dentro del cometido principal de la presente memoria de tesis doctoral, es el de proporcional una herramienta de especificación y validación de modelos bioinspirados de visión. Dicha herramienta contiene a su vez distintas utilidades específicas que permiten entre otras cosas:
  - Visualizar y analizar los datos biológicos que se quieren mimetizar.
  - Generar patrones para campos receptivos definidos por el usuario.
  - Actualizar el software por medio de internet.
- *Módulo de modelado en hardware reconfigurable*: *CodeSimulink*, es el entorno de trabajo escogido para definir digitalmente el modelo funcional de visión descrito por *Retiner*. Se basa en el lenguaje de descripción de hardware *VHDL*. El capítulo 6 muestra la batería de módulos y primitivas hardware diseñadas e integradas con *CodeSimulink* que han sido escritas

para conformar el modelo de visión. Este entorno, además de una cuidada independencia de plataforma, proporciona además una independencia con distintas herramientas comerciales de síntesis de sistemas digitales. La adaptación de *CodeSimulink* para que procese modelos de visión ha sido otro de los hitos de la presente tesis.

- *Módulo de síntesis automática de alto nivel*: Descrito en el capítulo 7, este módulo se encarga de trasladar a una representación digital susceptible de ser sintetizada en hardware, la familia de modelos de visión que pueden ser definidos por *Retiner*. El entorno, cuyo nombre es *HSM* [29, 30, 31, 32], acrónimo de *Hardware Software Maker*, está compuesto por una serie de *APIs* y utilidades (analizadores de código, compiladores/traductores, optimizadores, etc...) que dotan a la plataforma informática propuesta de la habilidad para sintetizar en hardware reconfigurable, modelos de procesamiento de datos descritos de forma altamente funcional. Este módulo está también intrínsecamente ligado con el anterior, ya que *HSM* utiliza las primitivas hardware de éste para describir los modelos de forma digital.

4.2.2. Plataformas software y lenguajes empleados

El hardware empleado por dicha plataforma, está compuesto de una arquitectura informática concreta más una arquitectura reconfigurable tipo *FPGA*. La independencia de un cierto *software* de la arquitectura informática subyacente a toda plataforma informática, es un cometido de suma importancia en la actualidad. Entre los esfuerzos más destacados se encuentra la plataforma *Java* o a la reciente *plataforma Microsoft .Net*; ambos proyectos comandados por el paradigma de ejecución de código mediante una máquina virtual *Java* o *.Net* según el caso. Estos proyectos han puesto de manifiesto los beneficios que tiene un determinado sistema *software* de ser *multiplataforma*, término éste que se define a continuación:

Multiplataforma: Es la cualidad que tiene un determinado sistema *software* de ser ejecutado o fácilmente transportable a distintas plataformas, con independencia del sistema operativo y la arquitectura informática. Como ejemplo de dicha cualidad se puede citar:

- Las aplicaciones *Java* o *.Net* nativas. Son aplicaciones que pueden ejecutarse en todas las plataformas que soporten la máquina virtual oportuna en cada caso.
- Un código fuente en lenguage *C* que hace uso sólamente del *API* (*Application Programming Interface*) estándar del *C* (*ISO C90*, *ISO C99*, ...) y que por tanto es fácilmente transportable de una plataforma a otra mediante un compilador al efecto (por ejemplo *GNU gcc*).

Por tanto, se ha creido conveniente dotar a la plataforma que se propone, de una independencia con la arquitectura que se integra con ella, es decir, de la

cualidad de *multiplataforma*. De este modo, teóricamente podemos emplearla con distintos *sistemas operativos* como *Linux*, *Mac OS X*, *Sun Solaris*, *Microsoft Windows*, *(Free, Net, Open)BSD*, ...ejecutándose en arquitecturas del tipo *x86−64*, *IA64*, *x86*, *SPARC*, *PowerPC*, ...
En la práctica y tal como se comentan en los capítulos 5 a 7, este objetivo se ha conseguido con ciertas excepciones, que son convenientemente comentadas en dichos capítulos.
A continuación se muestra el elenco de paquetes software y lenguajes que utiliza la plataforma que nos ocupa:

- Lenguaje *Matlab*. Producto de *MathWorks*, es a la vez, un entorno destinado principalmente a la matemática discreta y el potente lenguaje de programación asociado. La sección 5.2.1 del capítulo 5 presenta de forma más detallada los detalles de este programa, del que ahora principalmente interesa su presencia en distinas plataformas informáticas, como *x86 + Windows o Linux* y la portabilidad entre ellas de los distintos programas realizados en lenguaje *Matlab*. Este lenguaje está presente en todos los módulos que componen la plataforma. Aproximadamente el $80\,\%$ de *Retiner* está escrito en este lenguaje. También, *Codesimulink* por su parte está mayoritariamente escrito en lenguaje *Matlab*.

- Los lenguajes *C* y *C++* han sido empleados para realizar la interfaz directa con el hardware reconfigurable y otros dispositivos de video disponibles en *Retiner*. Las utilidades más destacadas en este sentido son:

  - *Live Probe*: Una aplicación en *C++* para comprobar los modelos de visión generados por *HSM* y *CodeSimulink*, utilizando la entrada de una webcam o similar y la placa *RC1000* (que se presentará a continuación). *Live Probe* es capaz de configurar la *FPGA* y visualizar convenientemente el resultado de su procesamiento. Está programado utilizando la biblioteca *VCL* (*Borland's Visual Component Library*) y el compilador *Borland C++Builder* 6.

  - *Microelectrode Selection Tool*: Es una utilidad híbrida, escrita en lenguaje *Matlab* y en *C++*. Se trata de una utilidad para seleccionar convenientemente los electrodos a visualizar. La interfaz de selección del conjunto de microelectrodos está escrita en *C++* utilizando *Borland C++ Builder* y *VCL*, ya que el *API* de interfaz gráfico de *Matlab* no ofrece la potencia necesaria.

  - *HSM−RI Video Driver Interface*: Es una *DLL* de *Windows* diseñada para ser utilizada por *Matlab* como un *recubrimiento* de la biblioteca *Microsoft Video for Windows*. Esta *DLL* está basada en el proyecto *Video for Matlab* de código abierto, al que hemos añadido nuevas funciones. Esta interfaz con una videocámara no es portable y actualmente está sólo soportado en la plataforma que nos ocupa, para sistemas *Windows*.

- *awk*[33]: Este lenguaje ha sido utilizado para acelerar el procesamiento de los archivos biológicos de respuesta a estímulos. Estos archivos contienen una gran cantidad de información en modo texto que hay que filtrar y modelar convenientemente para poder ser procesada por *Matlab* y *Retiner*. Se comprobó que *Matlab* es bastante lento para leer y procesar un archivo de texto y se probaron distintas alternativas. Como curiosidad comentar que se intentaron hacer varias versiones del mismo programa en *C* y no se obtuvo mejora alguna de rendimiento con el script en *awk*. También se probaron otros lenguajes como *Perl*, lenguaje orientado precisamente al procesamiento masivo de información, saliendo siempre ganador *awk* para esta tarea concreta.

- *Perl*[34]: Acrónimo de *Practical Extraction and Report Language*, el lenguaje Perl ha sido empleado para extraer información y procesar rapidamente los archivos de trenes de impulsos generados por *Retiner* además de los análogos biológicos. *Matlab* tiene soporte nativo de *Perl* en la versión que se ha utilizado (6.5), por lo que no hay que instalar ningún componente adicional. Los programas *Perl* de la plataforma son *scripts* que ejecuta *Retiner* para realizar tareas de adecuación y procesamiento de estos datos y permiten a su vez la ejecución externa a *Matlab* por cualquier intérprete de *Perl*

- *Java*[35]: El lenguaje *Java* ha sido utilizado en las tareas de conexión a internet para actualizar el software o conocer las posibles actualizaciones. Java también es un lenguaje empotrado dentro de *Matlab*, por tanto, tampoco necesita ningún componente adicional para ejecutar las actualizaciones. También se ha utilizado para visualizar una pantalla tipo *splash* de bienvenida al programa *Retiner*.

- *C#*[36]: La principal aportación de la plataforma que se presenta, es el módulo de generación automática de sistemas digitales a partir de descripciones en alto nivel de sistemas de tipo flujo de datos o *dataflow*. Para la escritura de dicho módulo, se ha utilizado tanto el lenguaje *Matlab*, para interaccionar de forma nativa con *CodeSimulink* y *Matlab*, y el nuevo lenguaje *C#*. Por motivos de compatibilidad no se han utilizado los componentes y las bibliotecas no portables de este lenguaje. La sección 4.3 presenta el lenguaje *C#* y el entorno de ejecución escogido. La principal razón para la elección de este lenguaje es el rico *API* con que viene compañado, fruto del entorno de ejecución, su rapidez de ejecución y la condición de lenguaje multiplataforma similar a *Java*.

4.2.3. Plataforma hardware y lenguajes empleados

Como ya se ha comentado en el presenta capítulo, el lenguaje principal para la descripción de los módulos hardware de la plataforma informática es el lenguaje *VHDL*. Si bien, al principio del trabajo relativo a la presente memoria, y como se detalla en el capítulo 6, también se realizaron distintas pruebas,

implementando módulos análogos a los definitivos (en *VHDL*) con el lenguaje *Handel–C*.
Se introduce a continuación una breve reseña sobre el lenguaje *VHDL*:

VHDL :Se trata de un lenguaje fuertemente tipado y muy parecido al *Ada* que se ha convertido en un estándar en la industria. Existen múltiples simuladores y sintetizadores lógicos que utilizan este lenguaje como la entrada de diseño principal y que dada su madurez, se puede argumentar que también es el *HDL* que produce una síntesis actualmente más optimizable por dichas herramientas.

El estándar ha sido ampliado para incluir:

- Estensiones para circuitos analógicos y mixtos.
- *VITAL*. Iniciativa para diseñar circuitos *ASIC*.
- Extensiones para desarrollar circuitos de microondas.

Los factores anteriores han primado en la elección de este lenguaje para la plataforma.
Por otro lado, la plataforma hardware utilizada es, como se ha comentado, el hardware reconfigurable. Se ha procurado que los módulos hardware diseñados sean independientes del sintetizador lógico que los procese, y del tipo y modelo de *FPLD* (*Field Programmable Logic Array*) en el que se quiera proyectar. Concretamente, para la los primeros prototipos de sistemas de visión, se ha utilizado la placa de *AlphaData RC*–1000, que incluye una *FPGA VirtexE-2000*. El capítulo 6 presenta estos cometidos de una forma más extensa y precisa.

4.3. Elección de *C#* y Mono como plataforma software de desarrollo y ejecución.

C# es un lenguaje sencillo, moderno, orientado a objetos y de alto rendimiento con la plataforma *.Net* de *Microsoft* cuya definición se presenta a continuación:

La plataforma *Microsoft .Net* es en esencia un nuevo *API* multiplataforma para confeccionar programas que se ejecutan en la máquina virtual de *.Net* o *CLR* (Common Language Runtime). En reglas generales es un proyecto idéntico a *Java SDK*, salvo por el hecho de que la máquina virtual es independiente del lenguaje escogido, haciendo posible escribir un programa para *CLR* en lenguajes tan distintos como *Java*, *Fortran*, *Visual Basic*, *C++*, *Python*, etc. accediento todos ellos por igual a todas las funciones de la plataforma *.Net*, y generando un código *CIL* (Common Intermediate Language) que es análogo al *bytecode* de *Java* salvo por el hecho de que es imposible distinguir el lenguaje de alto nivel con el que ha sido generado. El lenguaje *C#*, la máquina virtual de *.Net* con su código intermedio

(CIL), así como gran parte del *API* de *.Net* están contemplados dentro de los estándares *ECMA (European association for standardising information and communication systems)*, *ISO (International Organization for Standardization)* *ECMA334*, *ECMA335* y *TR84* respectivamente.

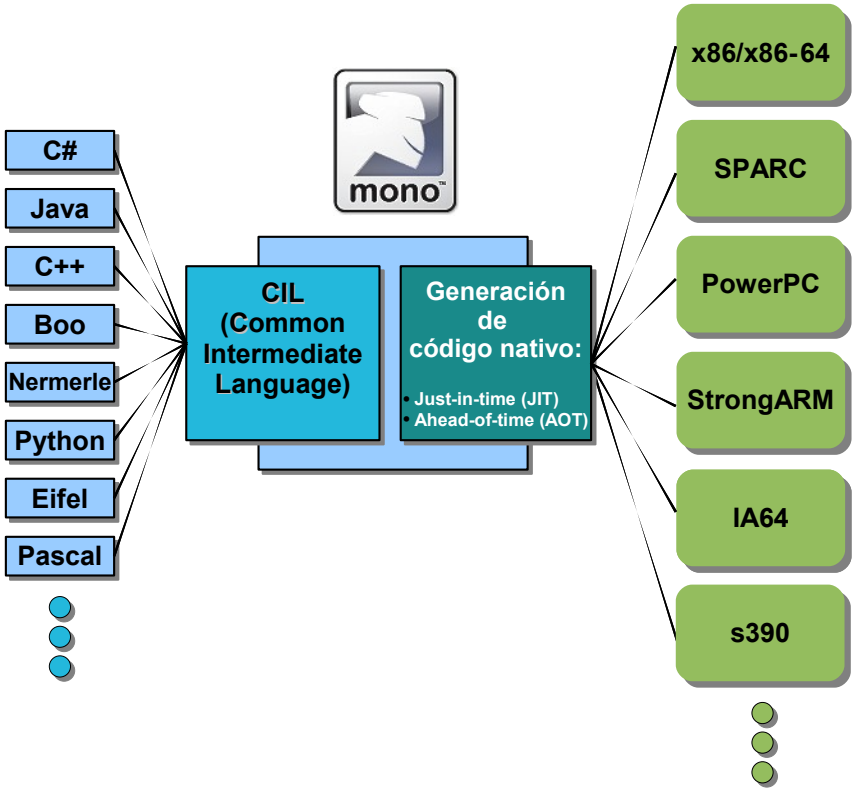

Figura 4.1: Esquema de la relacción del proyecto *Mono* con distintos lenguajes y plataformas hardware.

Basada en dicha plataforma, la empresa *Ximian*, actualmente integrada en *Novell*, ha desarrollado el *Proyecto Mono*[37], que pretende desarrollar los anteriores estándares utilizando licencias de software libre. De este modo, las licencias empleadas son las siguientes:

– *GPL o GNU General Public License* para el compilador de C#.

– *LGPL o GNU Library GPL 2.0* para el entorno de ejecución.

– *MIT X11* para la biblioteca de clases.

Aunque la plataforma *.Net* es independiente de la arquitectura, lo cierto es que *Microsoft* sólo da soporte para ella en sistemas *Windows* y por tanto sólo en

aquellas arquitecturas que lo soportan. Por el contrario, *Mono* es un proyecto multiplataforma y multilenguaje, tal y como se observa en el esquema[1] de la figura 4.1. Tiene compiladores para más lenguajes que la plataforma *.Net* (inicialmente con soporte sólo para *C#*, *C++*, *JavaScript* y *Visual Basic*) y soporta plataformas hardware muy dispares.
Es especialmente interesante, de cara a las líneas futuras de investigación que han nacido del presente trabajo de tesis doctoral (consultar los capítulos 8 y 9), el hecho de que al escribir una función o una clase en cualquiera de los lenguajes soportados por *.Net* o *Mono*, mediante la compilación, éstos quedan inmediatamente expuestos para ser utilizados por cualquiera de los demás lenguajes. Es decir, no hay necesidad de crear envolturas o *wrappers* de funciones o clases escritas en un cierto lenguaje, para poder ser reutilizables en otro.
Todas estas características han llevado a elegir al lenguaje *C#* y al *Proyecto Mono* como lenguaje principal y plataforma de ejecución respectivamente, para diseñar el tercer módulo.

4.4. Conclusiones

En este capítulo se ha introducido la plataforma informática de especificación, comprobación, validación y síntesis digital de sistemas de cómputo orientados al flujo de datos, articulándose ésta en tres módulos fundamentales: *simulación*, *modelado hardware* y *síntesis automática*. Se han definido a su vez, algunos conceptos y terminologías que se van a emplear en el resto de la memoria con objeto de clarificar y hacer más fácil su lectura. También se ha hecho un recorrido por las estrategias de programación empleadas en el proceso de diseño y escritura de la plataforma.
Las distintas herramientas y lenguajes que se han empleado para el diseño de la plataforma, han reforzado su carácter *multiplataforma* a la vez que beneficiado cada una de las tareas de síntesis que la integran:

- *Plataforma software*:

  $\rightsquigarrow$ *Modelado y simulación del sistema visual*: Maximizando con *Matlab* (tanto la aplicación como el lenguaje) la potencia de simulación y la interacción con las distintas aplicaciones de *CodeSimulink*.

  $\rightsquigarrow$ *Generación automática en hardware reconfigurable (Optimización y síntesis lógica)*: Mediante el empleo de *C#* y la tecnología *.Net/Mono*, se ha conseguido una interacción modular a las tareas de síntesis de alto nivel, a la vez que se ha mantenido el carácter de *multiplataforma* por medio de una interfaz transparente con lo módulos de simulación software (*Simulink*) y hardware (*CodeSimulink*). Los algoritmos de traslación software$\longrightarrow$hardware y optimización en área o velocidad, se ven beneficiados de su implementación usando la tecnología *.Net/Mono*, mediante las estructuras de datos, los algoritmos de alto nivel propios de la *OOP Programación*

[1] Imagen adaptada tomada de la documentación del proyecto Mono[38].

Orientada a Objeto (*Object Oriented Programming*) y sus posibilidades de mejora y/o edición con independencia de otras partes de la plataforma.

- *Plataforma hardware* La elección de *CodeSimulink* y el lenguaje *VHDL* como nexo de unión entre las tareas de simulación y síntesis de la plataforma, ha beneficiado a la portabilidad de los diseños hardware a la vez que a la interacción entre todas las herramientas constituyentes de la plataforma.

Capítulo 5

Simulación funcional de modelos de visión

El presente capítulo presenta y describe el módulo de especificación, simulación y validación de sistemas de visión bioinspirados bautizado como Retiner. *Tras una sección de introducción (5.1), la sección 5.2 introduce la aplicación* Retiner, *definiendo la plataforma software con la que ha sido escrito y fundamentando su elección. También se plantea el esquema general de uso que sigue dicho programa y su paradigma de especificación de alto nivel. Seguidamente la sección C.2 realiza un recorrido por todas las características fundamentales de la aplicación* Retiner, *mostrando a su vez la articulación de un sesión normal de diseño con esta herramienta. Finalmente la sección 5.3 presenta algunas experiencias de tipo psico–físico que han sido desarrolladas con* Retiner, *mostrando el carácter de multidisciplinariedad del dicho proyecto. El capítulo concluye con la sección 5.4 a modo de resumen.*

5.1. Introducción

UNa parte fundamental del presente trabajo de tesis doctoral, ha consistido en la implementación de una parte de la plataforma software de experimentación, comprobación y validación de modelos generales de visión dados en un alto nivel de especificación. Es decir, el primero de los módulos que integran la plataforma informática definida en el capítulo 4.1. Dicha plataforma ha sido desarrollada en el seno del proyecto europeo *CORTIVIS* con referencia *QLK6–CT–2001–00279*, que se ha presentado en la sección 3.5 del capítulo 3. El presente capítulo muestra las características fundamentales de dicha plataforma y profundiza en las estrategias que se han llevado a cabo para su diseño.

5.2. Simulación funcional de alto nivel: Retiner

La plataforma software de experimentación, comprobación y validación de modelos generales de visión definidos en un alto nivel de especificación, que se ha diseñado en *CORTIVIS*, tiene por nombre *Retiner*.

Frente a la lista de posibles alternativas en el mercado para llevar a cabo algunas o la mayor parte de las tareas que encierra dicha experimentación, comprobación y validación de modelos generales de visión, hemos optado por elaborar una aplicación propia que englobe todas las tareas en el marco de una solución de fácil manejo.

Nace así *Retiner*, como un esfuerzo para ofrecer en su cometido gran flexibilidad y potencia, a la vez que un uso sencillo y un marcado carácter multidisciplinar.

5.2.1. Plataforma software de desarrollo

La decisión más importante a la hora de diseñar y desarrollar *Retiner*, fue la de escoger un lenguaje apropiado para su implementación. El lenguaje inicial, que se escogió fue *Matlab*.

Matlab[39] es a la vez un entorno para la computación numérica y su lenguaje de programación homónimo. Es un programa comercial creado por *MathWorks* que permite un manejo extremadamente fácil de matrices, trazado de funciones y datos, implementación de algoritmos, creación de interfaces de usuario y la interación con módulos escritos en otros lenguajes. Aunque inicialmente pensado para el cálculo numérico, *Matlab* puede ser extendido por una serie de *ToolBoxes* o paquetes software que amplían de forma considerable sus capacidades. Concretamente, es posible usar matemática simbólica con el *ToolBox* adecuado. Uno de los paquetes software más interesantes de *Matlab* es la herramienta *Simulink*, que consiste en un motor de simulación de sistemas definidos mediante esquemas de bloques interconectados. Su utilización dentro de la industria y en entornos académicos está aumentando de forma significativa. Una de las características más notables de *Matlab* es que está presente en multiples plataformas hardware y sistemas operativos, incluyendo *Linux* y *Unix*, *Mac OS* y *Microsoft Windows*. Además, los programas escritos en lenguaje *Matlab* son portables entre las distintas arquitecturas y sistemas operativos que soporta *Matlab*.

Inicialmente, *Matlab* fue concebido como un recubrimiento (*wrapping*) de las conocidas bibliotecas para álgebra lineal y análisis numérico *LINPACK* y *EISPACK*. Estas bibliotecas, inicialmente escritas en *Fortran* estás presentes en la mayoría de las distribuciones de *Unix* actuales, incluyendo *Linux*. La idea original era poder manejar la potencia de las funciones para tratamiento de matrices que ofrecen estas bibliotecas, utilizando un nuevo y potente lenguaje lenguaje (*Matlab*), que evitase la tarea de escribir código en *Fortran*.

Actualmente existen clones de *Matlab* con licencias libres como la Licencia Pública General o *GPL*. Entre los más destacados encontramos a *Octave* o *Scilab*. Estos proyectos ofrecen una creciente compatibilidad con *Matlab*, a la vez

que una serie de características que no encontramos en la versión comercial.
En la actualidad, existe un nutrido grupo de paquetes *Matlab* que además de ser compatibles con *Octave* o *Scilab*, están protegidos con licencias libres. Y, de este modo, podemos encontrar, para casi cualquier disciplina académica, repositorios con paquetes software y funciones para estas aplicaciones.
Matlab se utiliza con éxito en las siguientes disciplinas (entre otras muchas): Ingeniería de Control, Procesamiento digital de Imágenes, Computación distribuida, Procesamiento digital de señales, Diseño de sistemas software/hardware, Optimización de funciones, Análisis de datos y Estadística, Comunicaciones, Análisis de modelos financieros. Concretando, las características que más destaco de *Matlab*, para defender su utilización en el desarrollo de *Retiner* son las siguientes:

- Amplio soporte en distintas arquitecturas (x86, SPARC, ...) y distintos sistemas operativos.
- Compatibilidad de los diseños creados entre las distintas arquitecturas y sistemas operativos.
- Ejecución rápida del código. *Matlab* está especialmente optimizado para efecturar operaciones entre matrices. La ejecución del código *Matlab*, que es interpretado, es aceptable comparada con la velocidad obtenida através de los lenguajes más tradicionales para este tipo de problemas como *Fortran*, *C* o *C++*.
- Las bondades y la sintaxis del lenguaje *Matlab*, especialmente en la escritura de funciones, es envidiable a la hora de ampliar, entender o interacionar con funciones escritas por el usuario en *Matlab*. La relación entre el costo de aprender el lenguaje y la potencia que podemos manejar es idónea.
- Existe una nutrida biblioteca de soluciones *Matlab*, con licencias libres o comerciales, que han facilitado enormemente la tarea de diseñar *Retiner*. Por ejemplo, el módulo de interfaz con un dispositivo de video utilizado.
- *Matlab* está integrado con otros lenguajes como *Perl* y *Java* dentro de la propia distribución de *Matlab*, con objeto de extender las posibilidades de éste. En la escritura de *Retiner*, se han usado estos dos lenguajes para permitir la actualización de *Retiner* via *Internet* (mediante una clase *Java*), y para acelerar enormemente el procesamiento sintáctico de datos (utilizando *Perl*). Más adelante en este capítulo, se concretará el papel de estos lenguajes.
- Posibilidades de interacción de *Matlab* con otros programas, via *Microsoft Dynamic Data Exchange (DDE)*, *sockets* de Internet TCP, empotramiento de componentes ActiveX y COM, y tuberías (*pipes*). En la elaboración de este trabajo de tesis doctoral, se ha experimentado con todas estas posibilidades. Finalmente, para tareas de interacción de *Retiner* con otras

aplicaciones, se ha utilizado tuberías o *sockets TCP* por compatibilidad con otras plataformas y sistemas operativos.

5.2.2. Retiner: Perspectiva general

La aplicación *Retiner* ha sido desarrollada en el marco del proyecto de investigación europeo *CORTIVIS*, con referencia (*QLK6–CT–2001–00279*) con objetivo de acometer el paquete de trabajo *WP2* de *CORTIVIS*, de nombre *Reconfigurable Bioinspired Visual Processing Front-end (artificial retina)*, correspondiente al hito D6 del proyecto: *Software implementation of a bioinspired model of the retina suitable for a cortical neurostimulation*

Como se comentó en el capítulo 1, el proyecto *CORTIVIS* está integrado por profesionales de areas tan distintas como la medicina (fisiólogos, histólogos, neurólogos, etc...), la ingeniería electrónica e informática, físicos, etc.... Es importante que la complejidad y la potencia de *Retiner* no venga en detrimento de la facilidad y accesibilidad de *Retiner* para la pluraridad de disciplinas que engloba *CORTIVIS*. Dadas estas premisas, que han decidido la forma de diseñar y desarrollar *Retiner*, podemos definir al mismo como:

***Retiner*:** Aplicación software desarrollada para *Matlab* y escrita principalmente en lenguaje *Matlab*. Sus objetivos principales son:

- Diseñar y comprobar mediante simulación, diferentes modelos de visión, haciéndo especial énfasis en las retinas artificiales. Incluye varios filtros espacio–temporales típicos en la composición de sistemas de visión bioinspirados.
- Validar el modelo de visión bioinspirado con los resultados obtenidos en la simulación en *Retiner* de dicho modelo y los datos biológicos correspondientes al mismo experimento.
- Facilitar diversas entradas para alimentar el sistema ed visión. Las disponibles hasta la fecha son:
  - Imágenes. En los formatos gráficos soportados por *Matlab*.
  - Videos. En los formatos de video soportados *Matlab*.
  - Entrada de video *on–line*. Utilizando un dispositivo de video compatible con *Microsoft Video for Windows* (avicap32.dll), por ejemplo, una sencilla webcam.
- El procesamiento del principal sistema de visión que nos ocupa, es decir, de un modelo de retina, dará como resultado la secuencia de eventos para enviar a la matriz de microelectrodos, tal y como se comenta en el capítulo 1.
- Facilitar la observación de la influencia que tienen los distintos parámetros de diseño de la neuroprótesis cortical, en la secuencia de eventos (matriz impulsos) que se enviará a la corteza visual.

- Facilitar la conformación del modelo de visión, así como todos los anteriores cometidos, gracias a una cuidada e intuitiva interfaz de usuario.

5.2.3. Requisitos recomendables para ejecutar *Retiner*

Retiner es un programa *Matlab*, con lo cual, los requisitos mínimos para su ejecución vienen impuestos por *Matlab*. En lo que respecta a los requesitos recomendables, con los cuales se ha obtenido una velocidad aceptable del motor de simulación de *Retiner* y del resto de sus utilidades, se destacan las siguientes reseñas:

- *Retiner* está diseñado en y para *Matlab Release 13 (6.5)*. Se ha comprobado su compatibilidad con las siguientes versiones de *Matlab*[1], si bien, como se comentará más adelante, existen algunos problemas técnicos dependientes del propio *Matlab*, que obligan a adaptar las versiones de *Retiner* para cada una de las nuevas versiones de *Matlab* con objeto de asegurar el mismo comportamiento en todas ellas.
- Un procesador de potencia igual o superior a la que ofrece un *Pentium IV* normal a 2 Gigahercios. A más potencia del procesador, más se acerca la velocidad de la simulación a una simulación rápida en *tiempo real*-
- Según *Matlab*, en cuanto al hardware para gráficos, es recomendable un adaptador gráfico de 24 o 32 bits (*true color*), a ser posible con aceleración *OpenGL*. Con las pruebas que se han efectuado con *Retiner*, se recomienda que la tarjeta gráfica no comparta su memoria con la memoria principal del sistema, ya que de este modo se ralentiza de forma visible el rendimiento de la simulación.
- Un dispositivo de video compatible con el sitema *Microsoft Video for Windows (avicap32.dll)*, en adelante *VfW*. Basta con una sencilla webcam. El formato de captura de las imágenes incide de forma decisiva en el rendimiento de la simulación on–line ya que suelen variar las dimensiones del *frame*.

5.2.4. Esquema general de *Retiner*

El esquema general en que está basado *Retiner* para diseñar sistemas bioinspirados de visión, se compone en líneas generales de un filtrado multicanal más una cierta codificación del mismo. Como su propio nombre sugiere, el esquema general de *Retiner* está orientado al diseño de retinas artificiales, que son un subconjunto de los esquemas generales de visión. Este hecho no viene en detrimento de que con *Retiner* sea también posible diseñar otros esquemas de visión más generales. *Retiner* por tanto, tiene una marcada vocación bioinspirada, con utilidades y aplicaciones específicas para este fin. El esquema o arquitectura general que utiliza *Retiner* puede consultarse en la figura 5.1.

[1] *Matlab Release 14 (7.0 y 7.1)*

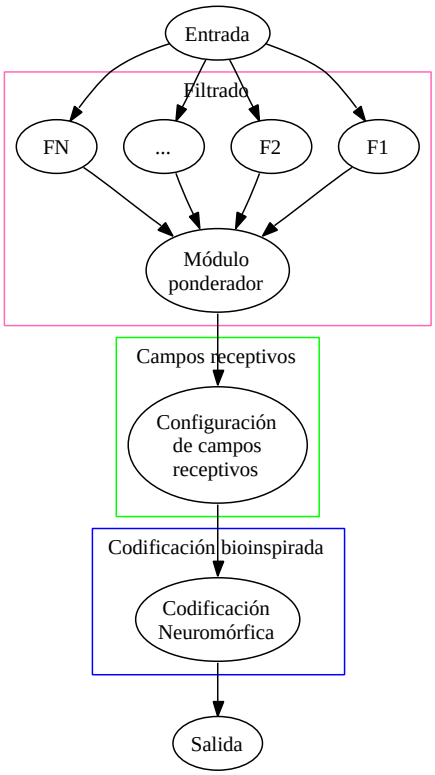

Figura 5.1: Arquitectura general del sistema de visión empleada por *Retiner*

En ella, la entrada al circuito de simulación puede ser cualquiera de las señaladas en el punto 5.2.2. Esta entrada será procesada por *Retiner* con objeto de diferenciar cada uno de sus canales cromáticos, que de forma habitual serán 3, el canal rojo (R), el canal verde (G) y el canal azul (B). De esta manera, y a diferencia de otras herramientas, y dado que tratamos con sistemas bioinspirados, los filtros por defecto de la biblioteca de funciones de *Retiner* permiten acceder de forma cómoda a cualquiera de los canales cromáticos de la entrada.

La entrada multicromática de *Retiner*, pasa por las siguientes etapas:

1. Alimenta a una serie de filtros espacio–temporales que son procesados en paralelo. El paralelismo al que nos referimos, motivado por la enorme granularidad que presentan los sitemas biológicos, sólo será efectivo en hardware. Esta es una de las causas que enlentecen la simulación de forma considerable, intensificando aún más las prestaciones en potencia del computador que realice la simulación. La salida de cada filtro proporciona la imagen filtrada por el mismo.

2. La salida de todos estos filtros, es ponderada según una sencilla ecuación

que debe introducir el usuario. Todos los filtros se nombran automáticamente por *Retiner*, desde `F1` hasta `FN`, y pueden emplearse en cualquier expresión normal de *Matlab*. La salida de este módulo, al que se ha nombrado como *módulo ponderador*, se llama *figura de información*, y es el resultado final de la simulación del modelo de visión. En un modelo de retina, la figura de información integra todas las características de la entrada (filtrado cromático, filtrado temporal, bordes, etc...) en una matriz de intensidad (que se hará visible posteriormente como una imagen en escala de grises).

3. Seguidamente, la figura de información es procesada por el módulo *Configuración de campos receptivos*, con objeto de determinar el área de acción que alimentará a un cierto electrodo. La contribución a cada electrodo es la media de todas las aportaciones que encierra su campo receptivo. Por defecto, el campo receptivo es rectangular y no hay solapamiento. La matriz que se obtiene en este paso, tiene las mismas dimensiones que la matriz de microelectrodos.

4. Por último, la salida de la etapa anterior se procesa con un módulo llamado *codificador neuromórfico*. Este módulo procesa la excitación enviada a cada electrodo mediante un modelo neuronal del tipo *integra y dispara* (*Integrate and Fire*), de forma que produce impulsos en el tiempo para cada electrodo. Estos impulsos valen 0 ó 1.

5.2.5. Descripción del modo de trabajo con *Retiner*

La figura 5.2 muestra la captura de pantalla que obtenemos al iniciar *Retiner*. Como puede observarse, la interfaz de usuario está diseñada para acceder fácilmente a todos los parámetros que conforman un modelo de visión bioinspirado. De esta forma, nos encontramos de izquierda a derecha con las siguientes entradas de menú:

1. Menú `Source`. En el que podemos elegir el tipo de entrada que tiene nuestro esquema, que como comentábamos en el punto 5.2.5, puede variar entre una imágen estática, archivo de video o captura en vivo de un dispositivo de video. El despliegue de este menú puede verse en la figura 5.3

2. Menú `Photoreceptors`. Al seleccionar este menú, podemos acceder al cuadro de diálogo que se muestra en la figura 5.4, en el que podemos variar la ganancia de los receptores. Los valores de dichas ganancias no tienen porqué ser literales, aceptándose una variable *Matlab* adecuada. El valor por defecto de la ganancia de cada canal cromático es 1.

3. Menú `Filtering`. En este menú es posible:

   - Añadir cualquiera de los filtros predefinidos de *Retiner*, accediendo de forma cómoda a la configuración de cada filtro mediante la edición de cada uno de sus parámetros.

- Añadir al entorno un conjunto de filtros previamente definidos en un archivo de filtros de *Retiner* (con extensión `.fil`).
- Guardar el conjunto de filtros actual
- Seleccionar una rectificación de media onda al resultado de aplicar la combinación de filtros indicada por el usuario (en la línea de edición `Combination`, de la ventana principal de *Retiner*).
- Podemos indicar si queremos visualizar el resultado de la mencionada combinación de los filtros.

4. Menú `Receptive fields`. Tal como muestra la figura 5.5, mediante esta entrada de menú tenemos acceso a la configuración del los campos receptivos de cada uno de los microelectrodos, así como a la edición de las dimensiones de éste. Por defecto, se toman las dimensiones matriz de microelectodos de la *Universidad de Utah* (10×10)[17], y se suponen campos receptivos cuadrados sin solapamiento mútuo. La lista completa de acciones a que da acceso este menú es:

   - `Activity matrix size ...`. Donde se pueden editar las dimensiones de la matriz de microelectodos (número de columnas y filas) que se supone reectangular.
   - `Receptive fields`. Submenú en el que se especifica si se quieren unos campos receptivos rectangulares y no solapados, o por el contrario, se quieren unos campos receptivos definidos por el usuario con una de las utilidades externas de *Retiner* (opción accesible mediante el submenú `Custom...`).
   - `Show Activity Matrix`. Submenú de tipo *check box* para indicar a *Retiner* si deseamos o no visualizar la matriz de actividad.
   - `Graded output method`. En el caso de que se desee mostrar la matriz de actividad, mediante este submenú, se puede escoger entre un suavizado gaussiano (el más próximo a la biología), bilineal o bicúbico. La visualización suavizada de la matriz de actividad es una primera aproximación a una reconstrucción de la imagen en el cerebro.

5. Menú `Electrode stimulation`. Con este submenú, se accede a la configuración de los parámetros del modelo neuronal de tipo *integra y dispara* que utiliza *Retiner* para conformar los trenes de implusos (*spikes*) correspondientes a cada electrodo. También se accede a una utilidad externa de *Retiner* para comparar trenes de impulsos. Se detallan a continuación, cada una de las entradas:

   - `Options for AVI processing...`. Si se ha seleccionado como entrada a *Retiner* un archivo de video, este submenú se activará para permitirnos fijar tres parámetros que ajustan la resolución temporal del archivo de video.

     Los parámetros a los que se tiene acceso son:

- `Time Step (ms)`. Que determina la frecuencia de muestreo a la que se procesará cada *frame* del archivo de video.
- `Temporal filter resolution (time steps)`. Es el número de pasos temporales que separan 2 *frames* consecutivos a considerar en los filtros de realce temporal. *Retiner* no permitirá un paso temporal más pequeño que la propia resolución del archivo de video.
- `Number of frames for averaging (at least 1)`. Es el número de frames correspondientes a cada paso de simulación, que se tendrán en cuenta para la posterior comparación con el frame actual.

6. Menú `Plugins`. El contenido de este menú se contempla específica y ampliamente en el capítulo 7 del presente texto.

7. Menú `?`. El submenú de ayuda y documentación `?`, da acceso a la documentación que acompaña a cada copia de *Retiner* y busca nuevas versiones del mismo. Este submenú da acceso a las siguientes opciones de menú:

   - `About...`. Que muestra los créditos del programa *Retiner*.
   - `Search for last version`. Esta opción utiliza una clase de *Java* (instalado por defecto en cada versión de *Matlab*) para buscar en *Internet* una versión de *Retiner* más actualizada. Después de cierto tiempo, esta opción nos dirá si tenemos la última versión de *Retiner* y eventualmente si hay otra versión más moderna de este. La figura 5.9 muestra el resultado de la aplicación de esta orden. La numeración de las distintas versiones de *Retiner*, sigue el esquema de numeración del kernel de *Linux*, es decir, el primer número es la rama principal de la versión, el segundo la versión de dicha rama y el tercer número la *release* o *versión pequeña*. Las distintas *release* indican cambios menores en *Retiner* y resolución de errores (*bugs*), los dos primeros números hacen referencia a cambios más importantes en el diseño de *Retiner*.
     Si la versión de la rama es un número impar, tal y como ocurre en todas las versiones del kernel de *Linux*, indicará que se trata de una versión de desarrollo potencialmente inestable y poco probada. Quedará al usuario la decisión de actualizar a esta nueva versión con su consiguiente riesgo.
     La clase de *Java* nos avisará convenientemente de la opción más aconsejable a seguir. En el ejemplo de la figura 5.9, observamos que tenemos una versión tipo beta (una candidata a versión final) con numeración 1.0.0.

5.3. Experiencia de multidisciplinariedad con *Retiner*

Uno de los tests psicofísicos más interesantes que se pueden llevar a cabo con *Retiner*, consiste en utilizar el modo *pantalla completa* para visualizar la salida, emborronada de forma gausiana, de la matriz de actividad correspondiente a un modelo bioinspirado de retina proyectando sobre una matriz de microelectrodos.

Figura 5.10: Imagen izquierda: Gafas de Rimax Virtual Vision 2.0. Imagen derecha: Gafas Sony Glasstron

Con esto se consigue simular la posible excitación cortical que se pretende llevar a cabo. El emborronamiento gaussiano de la señal de trenes de impulsos, salida final de *Retiner* es modelo bioinspirado de la difusión de esta señal en la corteza visual, concretamente en $V1$.

Conectando esta salida a unas gafas especiales de visión completa como las *Rimax Virtual Vision* 2.0 o las *Sony Glasstron* de la figura 5.10, se ha demostrado que es posible distinguir suficientemente el ambiente visual que rodea a un individuo, utilizando sólamente una matriz de 10×10, dimensiones estas de la matriz de microelectrodos de la Universidad de Utah [17] que se ha utilizado en el proyecto *CORTIVIS*.

Las gafas de *Rimax* tienen una resolución de 800×225 píxeles, con sonido integrado, un *display* visible de 36 pulgadas y varios conectores digitales, todo ello en 200 gramos. Las de *Sony* por otra parte tienen un display visible de 72 pulgadas, las demás características son similares.

Se han realizado los siguientes experimentos psicofísicos:

- Identificación de letras y palabras. Como mínimo las dimesiones de la matriz de microelectrodos deben ser de 10×10, las letras deben tener unas dimensiones comparables con el centímetro y la distancia cámara–letras suele rondar los 30 cm.
- Posibilidad de exploración satisfactoria de áreas abiertas.

Se ha comprobado, que para el correcto y más rápido reconocimiento de patrones, es muy útil observar el objeto en cuestión valiéndose de unos pequeños movimientos exploratorios relativos a su periferia. Esto está directamente relacionado con la referencia bibliográfica [40] (*Retinal ganglion cell synchronization by fixational eye movements improves feature stimation*) y los movimientos microsacádicos del globo ocular.

En la actualidad, están siendo proyectados distintos experimentos psico–físicos con *Retiner* con el objetivo de cuantificar más detalladamente las bondades del modelo retinal que se está utilizando y un futuro implante cortical.

5.4. Conclusiones

En el presente capítulo se ha presentado y descrito el módulo de especificación, simulación y validación de sistemas de visión bioinspirados bautizado como *Retiner*. Así mismo, se ha profundizado en su diseño y modo de diseño, haciendo hincapié en su carácter multidisciplinar. Finalmente se han presentado algunas experiencias de tipo psico–físico desarrolladas con *Retiner*, mostrando el carácter de multidisciplinariedad del dicho proyecto y se han focalizado nuevas líneas de investigación y trabajo con esta herramienta.

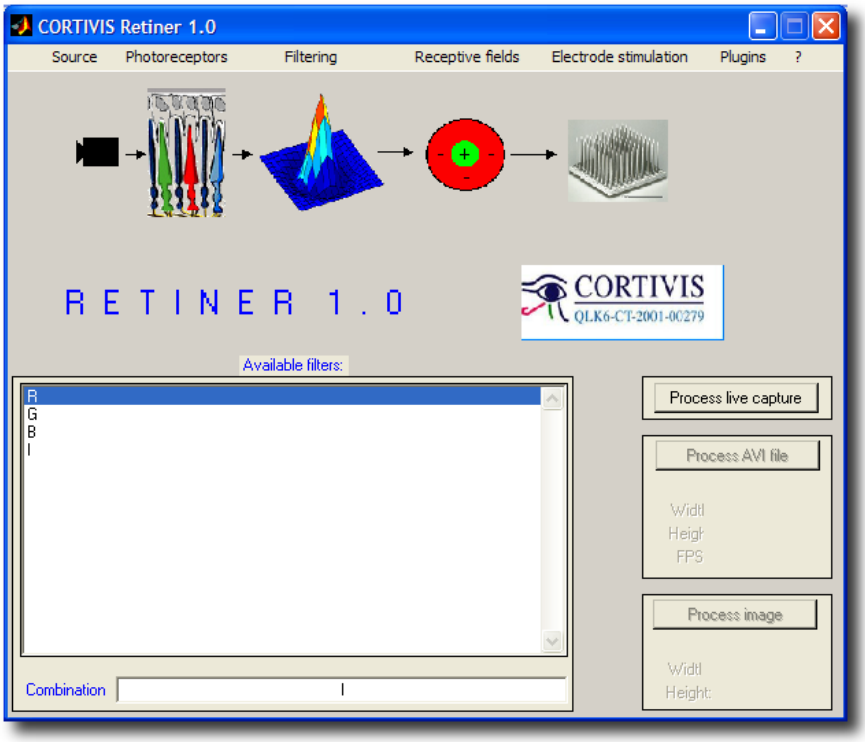

Figura 5.2: Captura de pantalla de *Retiner* al comenzar una sesión.

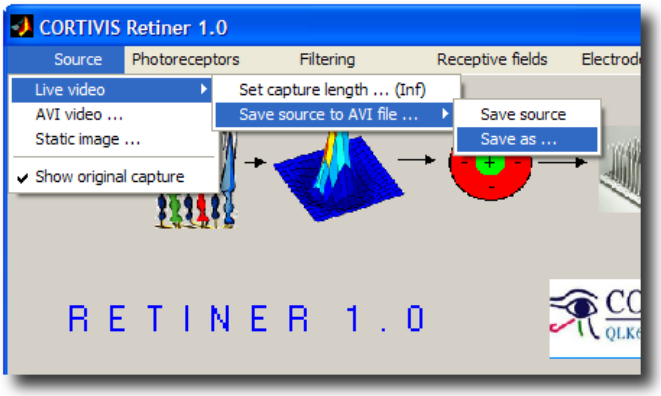

Figura 5.3: Menu `Source` de *Retiner*.

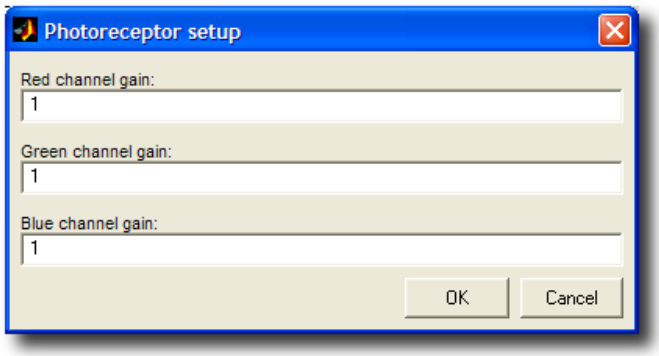

Figura 5.4: Cuadro de diálogo para editar la ganancia de los canales cromáticos.

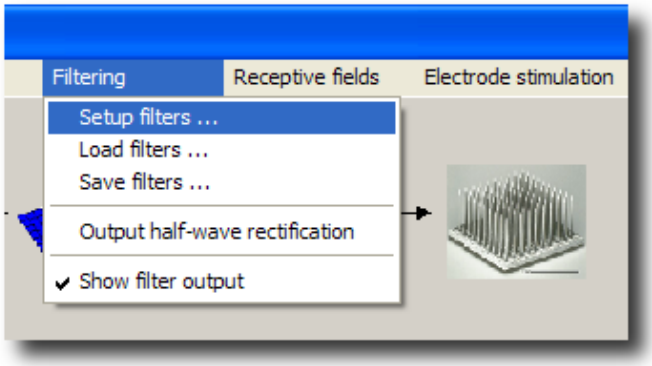

Figura 5.5: Menú para editar y definir los filtros.

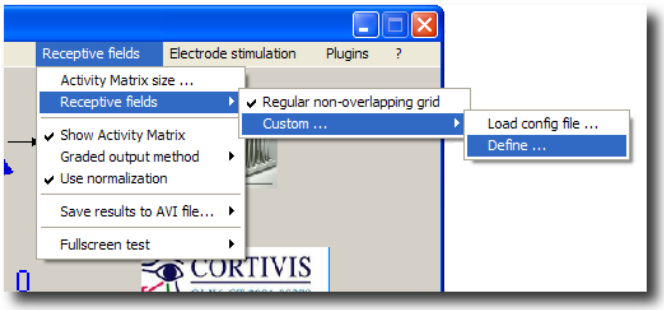

Figura 5.6: Menú para editar los campos receptivos.

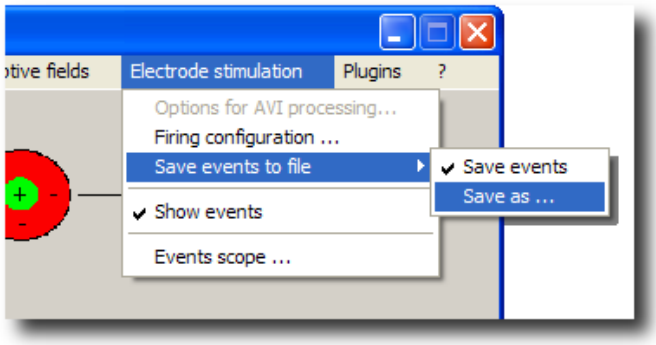

Figura 5.7: Menú para configurar el modelo neuronal tipo *integra y dispara*, del codificador neuromórfico.

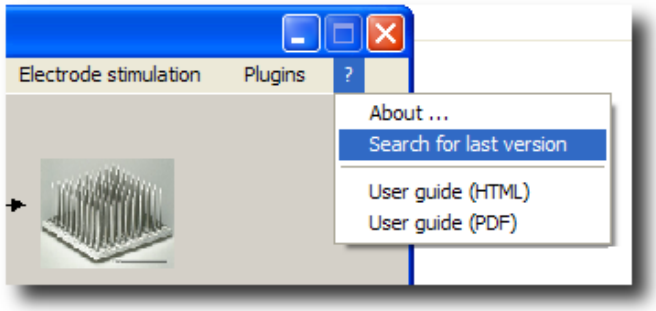

Figura 5.8: Menú de ayuda y documentación.

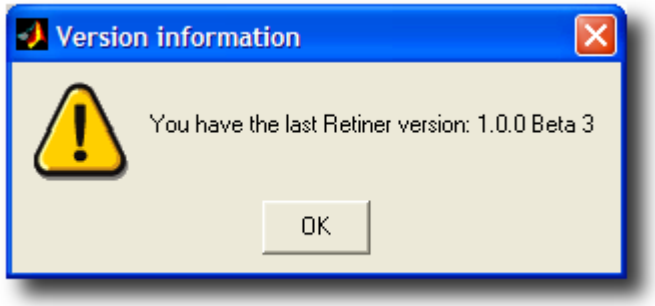

Figura 5.9: Cuadro de diálogo mostrando información sobre la versión actual que se tiene instalada de *Retiner*.

Capítulo 6

Implementación hardware

En el presente capítulo se presenta el Módulo de modelado en hardware reconfigurable *de la plataforma de diseño de sistemas de visión introducida en el capítulo 4. Se describen los módulos* VHDL*, así como los elementos de la interfaz de usuario necesarios para implementar dicho módulo. En la sección 6.1 se hace un análisis, a modo de introducción, de cómo los requisitos hardware de los sistemas de visión han influido a la hora de diseñar la plataforma y definir la arquitectura de referencia para el sistema de visión. La sección 6.2 comenta la elección del entorno de desarrollo* CodeSimulink–Matlab–Simulink *como base para elaborar la plataforma. Seguidamente, la sección 6.3 expone la propuesta en hardware reconfigutable para la realización del sistema de visión y presenta los nuevos bloques* VHDL*, compatibles con* CodeSimulink*, que se han escrito para las tareas de procesamiento de imágenes. Finalmente la sección 6.4 concluye el capítulo resumiendo su contenido.*

6.1. Introducción

El rendimiento requerido en sistemas de procesamiento de información visual, que generalmente necesitan desarrollos en *tiempo–real*, ha llevado a plantear el diseño de la plataforma hardware que nos ocupa en este trabajo según una implementación completa sobre *hardware reconfigurable*.

Además, las consignas del proyecto europeo *CORTIVIS*, en donde inicialmente comenzó y se fundamentó este trabajo de tesis doctoral, señalan también en la misma dirección en cuanto al uso de hardware reconfigurable. Como señalábamos en la sección 3.5 del capítulo 3, el proyecto *CORTIVIS* busca la realización de una neuroprótesis cortical para la restauración parcial de la visión en individuos con ceguera no congénita. Es decir, se trata de diseñar un dispositivo electrónico capaz de llevar a cabo los siguientes pasos de forma secuencial:

1. Capturar vídeo en tiempo–real a partir de uno o varios dispositivos de adquisición de imágenes (videocámaras, ...).

2. Procesar dicha información visual de forma parecida a como lo haría una retina biológica real. (Procesamiento retiniano de la información visual).

3. Codificar el resultado de dicho procesamiento de forma compatible con la biología neuro–cortical(*codificación neuromórfica*). .

4. Multiplexar, secuencializar y transmitir estas señales vía radiofrecuencia al circuito implantado en las primeras capas de la corteza visual de una persona invidente.

Por tanto, las exigencias de cómputo de los sistemas bioinspirados de visión, particularmente de una retina sintética, hacen destacar al hardware reconfigurable como la tecnología más idónea para desarrollar este trabajo doctoral.

También, a la hora de llevar a cabo el diseño de la plataforma en hardware orientada al procesamiento digital de imágenes, se han tenido en cuenta los siguientes aspectos:

- El tipo de sistemas de visión que queremos implementar, pueden ser especificados en base a un conjunto de filtros bien conocido y una relación matemática entre ellos. Arquitecturalmente esto genera una estructura muy marcada o dominada por el flujo de datos y con pocas rutinas de control.

- La plataforma hardware debe ser altamente parametrizable para adaptarse mejor a diferentes modelos de visión.

- Es recomendable no reducir la plataforma informática definida en el capítulo 4 a un tipo específico de hardware ni de software. Es decir, el diseño de la plataforma de ser compatible con:

  Distintas aplicaciones propietarias de síntesis y simulación de circuitos. Esto obliga a diseñar nuestro sistema primando la compatibilidad con distintas herramientas de síntesis lógica.

  Diferentes tecnologías propietarias compatibles con la plataforma hardware.

- Facilidad en la especificación en alto nivel. Que posibilitará la transparencia del diseño y lo abrirá a colectivos no directamente relacionados con el diseño de hardware.

- Posibilidad de simulación funcional de especificaciones en alto nivel de abstracción, para validar los diferentes modelos de visión. Esta simulación no debe ser en detrimento de la posibilidad de la simulación usual en *HDL*.

Todas estas características hacen decididamente recomendable el uso de una plataforma en hardware reconfigurable; sin embargo, esta elección no viene en detrimento de que el objetivo final de diseño sea o bien una tecnología

de hardware reconfigurable (FPLD) o una tecnología *ASIC* basada en celdas estándar.

La figura 6.1 describe a grandes rasgos el flujo de diseño que aconsejan los requisitos y características para la implementación del sistema visual en un circuito reconfigurable (*FPGA*). Nos encontramos con dos espacios de diseño, a los que me referiré según sus acrónimos en inglés, el espacio *VHLS* (*Very High Level Specification*) o *Espacio de Especificación de Muy Alto Nivel*, y el *HLS* (*High Level Specification*) o *Espacio de Especificación de Alto Nivel*. El espacio *HLS* engloba las descripciones de sistemas dadas en formatos *HDL* (*VHDL*, *Verilog*, *System–C*, *Handel–C* ...). Es en este nivel (*RT* o de *transferencia entre registros*) donde trabajan los sintetizadores lógicos comerciales, las herramientas de *Place&Route* y los simuladores de *HDL*. El espacio más interesante para la presente memoria es el *VHLS*. Aquí caben descripciones del sistema a muy alto nivel tales como:

- Modelos en lenguaje *Matlab*.
- Modelos en *HDLs* avanzados *System–C*, *Handel–C*, *System Verilog*, ...
- Modelos esquemáticos de alto nivel, como los que se encuentran en *CodeSimulink*, *System Generator*, *Altera DSP Builder*, *Celoxica PixelStreams* [41], etc...

El espacio *VHLS* está orientado a la simulación y validación funcional[1] de los modelos de visión.

Conectando el espacio *VHLS* con el *HLS* de forma apropiada, se puede llegar de una descripción funcional a muy alto nivel de un sistema, hasta una descripción *HDL* sintetizable mediante herramientas convencionales. Como veremos en el capítulo 7, el espacio *VHLS* permite una serie de optimizaciones del modelo para minimizar área o velocidad que constituyen, al entender del doctorando, uno de los resultados más relevantes de esta tesis.

Se ha escogido *VHDL* como lenguaje *HDL* para describir la plataforma de visión dentro del espacio *HLS*. También, por motivos que se justifican en la siguiente sección, se ha elegido la biblioteca *VHDL* de *CodeSimulink*, para sustentar y dar más posibilidades a nuestra plataforma.

6.2. Elección del entorno *CodeSimulink*

La idea de definir hardware mediante un esquemático de bloques complejos interconectados que encierran pequeñas unidades atómicas ya comprobadas y compatibles con simulación y síntesis, es un fenómeno relativamente novedoso en la industria actual. Con tales prestaciones, podíamos hablar de *System Generator* de *Xilinx*, *PixelStreams* de *Celoxica* y *Altera DSP Builder* entre otros [2].

[1] Frente al espacio *HLS*, cuyo objetivo es la síntesis lógica.
[2] consulte el capítulo 2 para más información

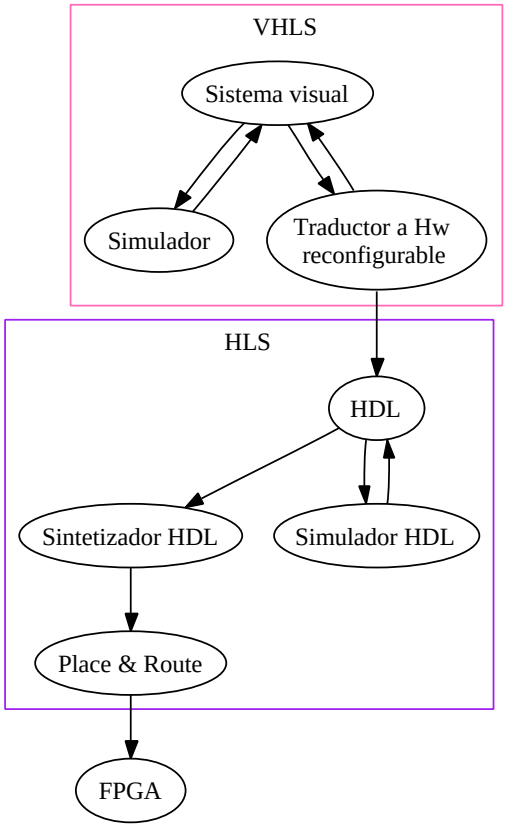

Figura 6.1: Primera aproximación al flujo de diseño de nuestra herramienta.

El entorno *CodeSimulink*, introducido en el capítulo 2, nos brinda la oportunidad de aunar gran parte de los objetivos deseados para diseñar nuestra plataforma, concretamente destaco:

- La especificación de sistemas hardware–software en alto nivel. En efecto, *CodeSimulink*, está basado en *Simulink*, y hereda de él todas las características de modelado de sistemas, en concreto su entrada de diseño conforme a un *esquemático*.
- Simulación de alto nivel. *CodeSimulink* permite, a través de *Simulink*, simular y *co–simular* modelos hardware/software definidos mediante bloques interconectados. En cada modelo se pueden incluir la mayoría de los bloques de *Simulink*, que aunque no son sintetizables, amplían considerablemente las posibilidades de simulación y visualización de resultados del modelo concreto.
- Integración en *Matlab* y *CodeSimulink*. *CodeSimulink* está escrito casi en su

totalidad en lenguaje *Matlab*, y es compatible con él. La interacción con *Matlab* es total y ofrece una forma potente, ágil, transparente y sencilla de acceder a todos los parámetros del diseño y de validación. Hago especial hincapié en la interacción con el proceso de simulación que vía *Simulink* permite visualizar y transformar los datos de una simulación incluso *on–line*.

- Por último, aunque quizás sea uno de los objetivos o características más interesante según mi criterio, el acceso al código fuente de *CodeSimulink* y la inestimable ayuda de su desarrollador principal, el profesor Leonardo María Reyneri, del *Politecnico di Torino*, para entender el complejo funcionamiento de esta herramienta.

La potencia de *Matlab*, entorno operativo en el que se basa *CodeSimulink*, ofrece un potentísimo abanico de posibilidades para crear una interfaz entre el espacio *HLS* y el *VHLS*. La herencia de las bibliotecas del *API*(*Application Program Interface*) de *Matlab* y *CodeSimulink* dotan a este binomio de las características necesarias para el desarrollo de la plataforma visual que nos ocupa.
Tal y como refleja la figura 6.2, *HSM* soporta los siguientes tres tipos de arquitecturas *CodeSimulink*:

- `parallelMatrix`: Donde todos los componentes de la unidad de procesamiento de datos (que en visión es una matriz bidimensional), se procesan en un sólo ciclo de reloj. La matriz de la figura 6.2 tiene dimensión 3×3, luego en un ciclo de reloj, se procesarán los 9 elementos de que consta.

- `parallelSerialMatrix`: En cada ciclo de reloj se procesa una columna de la matriz. En el ejemplo, cada 3 ciclos se procesa la matriz completa.

- `matrix` Donde en cada ciclo se procesa tan sólo una componente de la matriz, por lo que en el ejemplo, la matriz 3×3 tardaría 9 ciclos en procesarse.

A continuación se muestran los valores para los tipos de datos que usa *HSM* (que constituyen un subconjunto de los que usa *CodeSimulink*), ya presentados en el 2:

- *HSM_CSHI*: Acrónimo de *HSM CodeSimulink Hardware Implementation*, es un subconjunto de las posibles implementaciones *CodeSimulink* y que comentábamos antes, puede tener los valores {`parallelMatrix`, `parallelSerialMatrix` y `matrix`}.

- *CS_OV*: Acrónimo de *CodeSimulink Overflow*. Que expresa qué se debe hacer en caso de que el dato no sea representable en las condiciones impuestas por el usuario (longitud de palabra, posición punto decimal, etc...). Sus posibles valores son {Saturation, Wraparound y Back–propagated}.

- *CS_RO*: Acrónimo de *CodeSimulink Rounding*. Con el método de redondeo. Sus posibles valores son {Ceil, Floor, Round, Fix, Back–propagated}

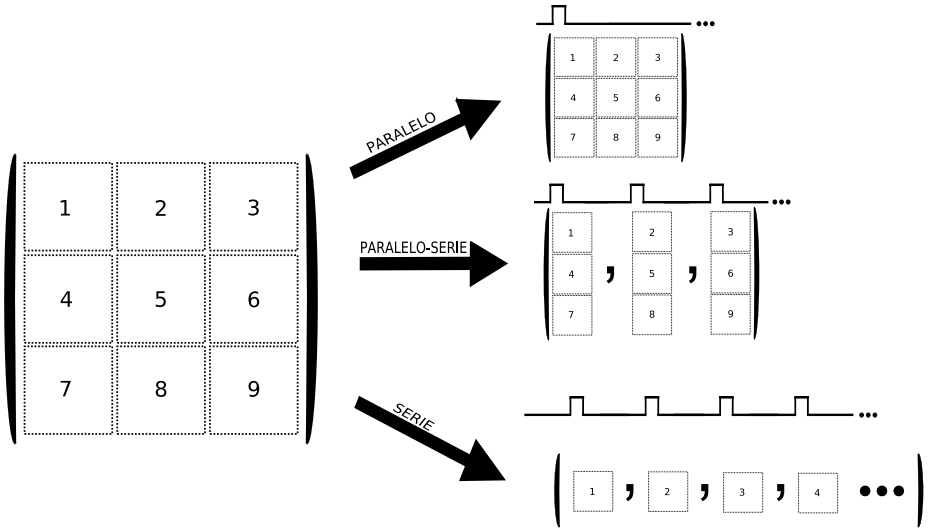

Figura 6.2: Esquemas de cómputo *CodeSimulink* soportados por *HSM*

- *CS_SR*: Acrónimo de *CodeSimulink Signal Representation*. Cuyos valores pueden ser {Unsigned, Signed, Modulus and sign, Float, Normalized float, Back–propagated}
- *natural*: Número natural ($\{1 \ldots \infty\}$).
- $natural_0$: Número natural o cero ($\{0..\infty\}$).
- *real*: Número real.
- *entero*: Número entero ($\{-\infty \ldots + \infty\}$).

A continuación se detallan los componentes *VHDL*, compatibles con la biblioteca *CodeSimulink*, que se han desarrollado para la plataforma visual.

6.3. Propuesta de sistema de procesamiento visual en hardware

Como se ha comentado en la sección 6.1, el tipo de sistema visual de que trata la presente memoria, se configura en torno a un conjunto de filtros espacio–temporales, debidamente ponderados y parametrizados según una ecuación matemática sencilla. La figura 6.3 muestra un esquema general de cómo se disponen los distintos filtros para conformar una sistema de visión compatible con la plataforma de visión que nos ocupa. En ella se puede observar cómo a partir de un flujo de datos de entrada (que vendrán suministrados generalmente por una videocámara a través de una memoria *RAM*), se procesan en paralelo n

filtros espacio–temporales, que serán oportunamente ponderados por el bloque de combinación lineal para dar el resultado final. Como ya se comentó en el capítulo 4, *HSM* es un conjunto de bibliotecas, programas y estrategias que han servido para llevar a cabo la generación automática de sistemas de visión en hardware reconfigurable, objeto de esta tesis.

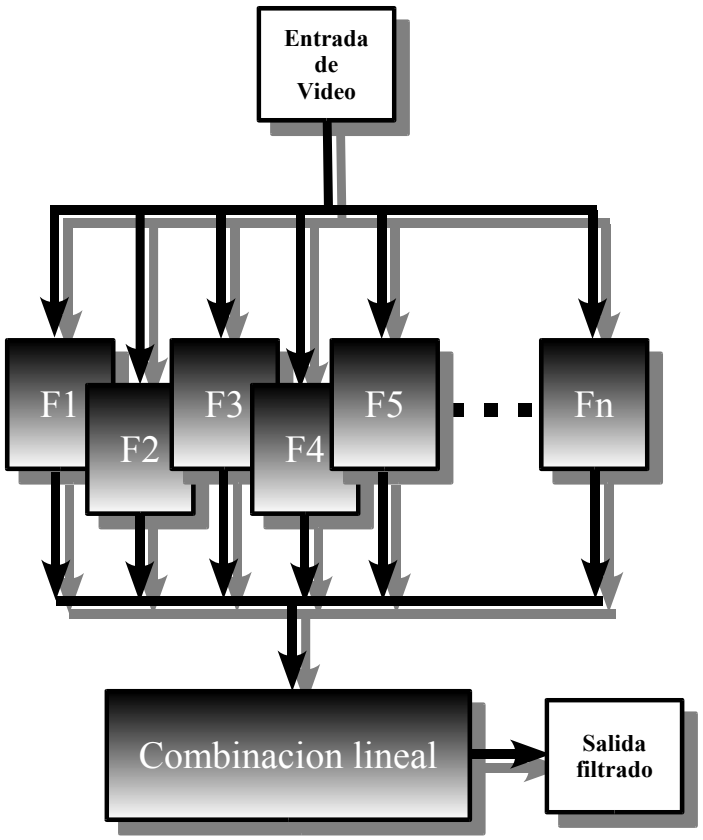

Figura 6.3: Esquema general de filtrado seguido.

6.3.1. Nuevos bloques añadidos a CodeSimulink

La compatibilidad con *CodeSimulink* de un circuito definido en *VHDL* implica que su estructura esté dividida en dos partes. La primera llevará a cabo el procesamiento del bloque en cuestión (sumador, comparador, filtro digital, etc...) y la segunda utilizará la interfaz con el protocolo *CodeSimulink* (consúlte la sección 2.4.1 del capítulo 2), que hará compatible al módulo con el resto de

recursos hardware de la biblioteca de *CodeSimulink*. La figura 6.4 muestra un esquema del diseño que se ha tenido en cuenta para implementar los nuevos bloques añadidos a *CodeSimulink*.

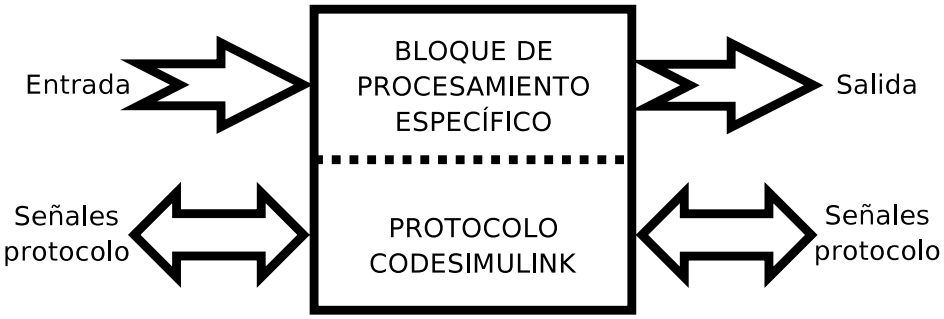

Figura 6.4: Esquema de un bloque *CodeSimulink*

6.3.2. El bloque *convolver*

Una de las primitivas más estrechamente relacionadas con el procesamiento digital de imágenes es la función conocida como convolución [42]. Se trata de un filtro *FIR* bidimensional que toma una matriz m de tamaño $M \times N$, un núcleo o kernel de convolución dado por una matriz k de tamaño $F \times C$ y opera según la definición dada por la ecuación 6.1 produciendo una nueva matriz $Conv$ de tamaño $M \times N$. Consulte la sección A.4.3 del apéndice A para empliar las distintas definiciones matemáticas.

$$Conv_{k,m}(x,y) = k(x,y) * m(x,y) = \sum_{i=-\frac{[F]}{2}}^{\frac{[F]}{2}} \sum_{j=-\frac{[C]}{2}}^{\frac{[C]}{2}} k(i,j) \cdot m(x-i, y-j) \quad (6.1)$$

para $x \in \{0 \dots M\}$ e $y \in \{0 \dots N\}$
Si la matriz m guarda la información relativa a los píxeles de una imagen dada, la aplicación literal de esta definición rota el kernel 180^o de forma que sólo los kernels con simetría polar generarían un resultado correcto. Normalmente en el campo del procesamiento digital de imágenes, conviene usar la función de *correlación* descrita en la ecuación 6.2. Además, es muy común ver bibliografía sobre procesamiento digital de imágenes donde se utiliza la palabra *convolución* para expresar realmente una *correlación bidimensional*. Este es el motivo de que nuestra convolución sea en realidad una correlación. No obstante, transformar una definición en otra es sólo cuestión de rotar la máscara del filtro 180^o. En adelante, me referiré solamente a la función de convolución, dada en la ecuación 6.2.

$$Corr_{k,m}(x,y) = k(x,y) * m(x,y) = \sum_{i=-\frac{[F]}{2}}^{\frac{[F]}{2}} \sum_{j=-\frac{[C]}{2}}^{\frac{[C]}{2}} k(x,y) \cdot m(x+i, y+j) \quad (6.2)$$

La ejecución de una convolución con un cierto kernel sobre una imagen, sigue la dinámica que muestra la figura 6.5. En ella se observa cómo una máscara de convolución 3×3 se va sucesivamente desplazando a lo largo de una matriz de imagen, desde la esquina superior izquierda, hasta la esquina inferior derecha de la imagen, de izquierda a derecha y de arriba hacia abajo. La sombra que produce la máscara sobre los 9 píxeles de la imagen que se representan debajo, señala el campo de acción del píxel central que se está procesando en ese punto.

La primitiva de convolución escrita para *CodeSimulink* se llama *sim_convolver_synchpar* y su código fuente puede consultarse en el archivo `sim_convolver_synchpar.vhd` de la distribución de *CodeSimulink*, y en el presente texto, en el Apéndice C.

El componente *sim_convolver_synchpar* está estructurado en varias unidades funcionales *HDL*.

En la figura 6.6 se muestra un esquema simplificado del funcionamiento interno del núcleo de convolución que se ha escrito especialmente para ser usado por *CodeSimulink*.

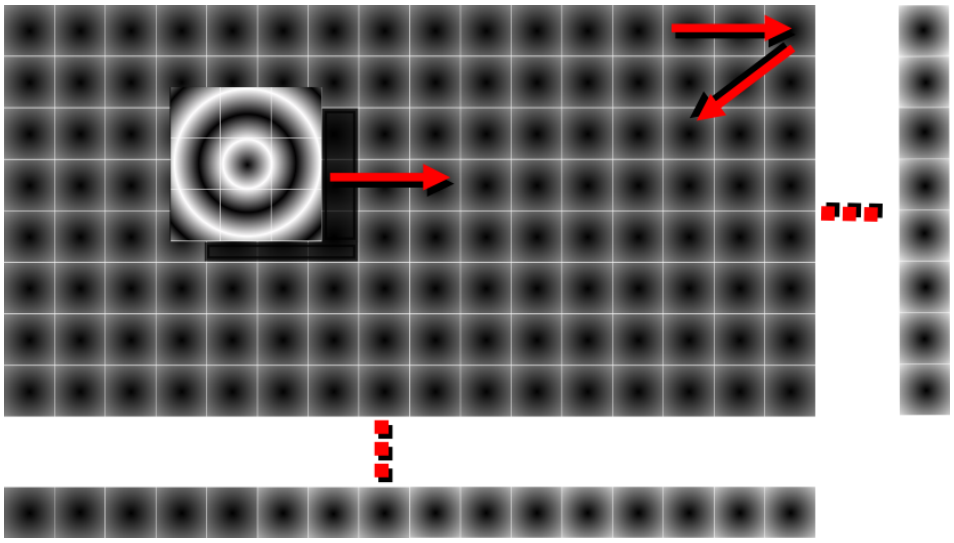

Figura 6.5: Esquema simplificado de la aplicación del módulo *convolver*

sim_convolver, que como todo bloque de *CodeSimulink* tiene una estructura hardware orientada al flujo de datos, calcula la convolución siguiendo secuencialmente los pasos:

1. Registro de los píxeles en una *caché* de tipo *FIFO* (*First data In, First data Out*). La longitud de la *FIFO* es un parámetro que puede tener longitud cero. Esta memoria ofrece robustez frente a posibles asincronías en los datos de entrada del módulo convolver. Por ejemplo, el módulo de convolución podría tomar su entrada de un *buffer* de vídeo conectado a una cámara. De este modo, aunque la cámara deje de funcionar durante algunos ciclos, el sistema total no se verá afectado, siempre y cuando el número de ciclos sea menor que el número de píxeles guardados en la memoria *FIFO*.

2. Registro de los píxeles en una estructura de registros de desplazamiento análoga a la propuesta de *Ridgeway* (ver [43]). Esta estructura genera la misma dinámica de desplazamiento de la máscara de convolución que la señalada en la figura 6.5 y selecciona adecuadamente los píxeles que se envían para su posterior multiplicación.

3. Multiplexación y registro de la salida para conformar el tipo unidad. Esta estructura es la que más se ve afectada del tipo de implementación hardware escogida. Concretamente el área ocupada por el multiplexor.

4. Multiplicación de los coeficientes de la máscara por los píxeles visibles por la anterior etapa. Nótese que dependiendo del tipo de datos *Code-Simulink* que se utilicen (matriz paralela, matriz paralela-serie o matriz serie), el módulo multiplicador implementará la multiplicación en paralelo, paralelo-serie o serie. Como ejemplo, si se tiene una máscara de convolución de dimensión 7×7, el módulo multiplicación actuando con datos paralelo-serie implementará 7 multiplicadores en paralelo, que llevarán a cabo la multiplicación en 7 ciclos de reloj. Si la matriz fuera serie, se implementaría tan sólo un multiplicador que computaría el resultado en $7 \cdot 7 = 49$ ciclos de reloj. Por último, una matriz paralela generaría 49 multiplicadores actuando en paralelo.

La figura 6.7 muestra el símbolo asociado al módulo *convolver*, cuyos parámetros de configuración pueden consultarse en la tabla 6.1.

Figura 6.7: Símbolo del bloque sim_convolver.

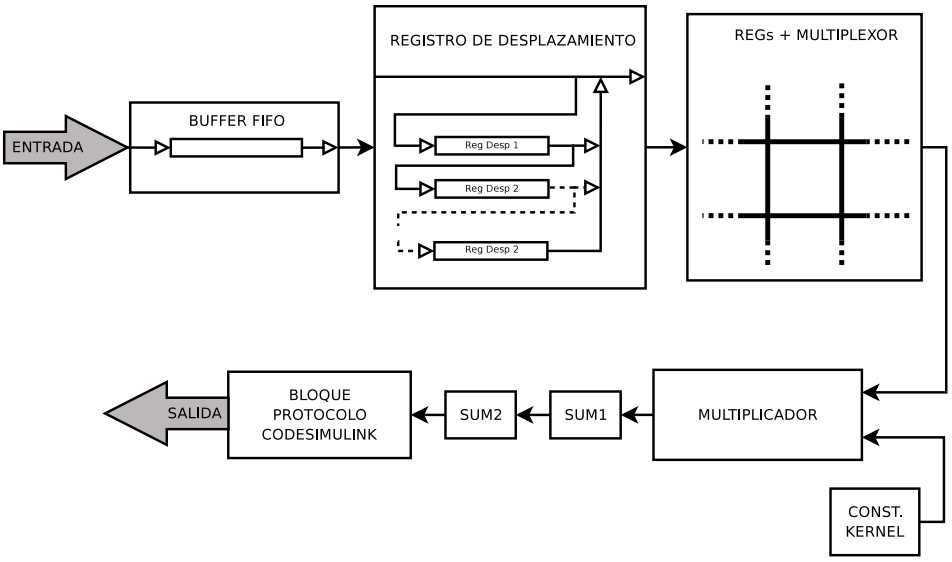

Figura 6.6: Esquema simplificado del módulo *convolver*

6.3.3. Bloques de interfaz

Interfaz con memoria externa

Los prototipos de la plataforma de visión se han desarrollado sobre la tarjeta aceleradora *RC–1000* de *AlphaData*, una tarjeta *PCI* que integra la *FPGA VirtexE 2000 BG560* de *Xilinx*. Esta placa incluye 4 módulos de *SRAM* externa a la *FPGA*.

Con objeto de poder usar estos recursos en la plataforma de visión que nos ocupa en la presente memoria, se han escrito 2 módulos *VHDL* de interfaz de la *FPGA* con un *"Módulo genérico de memoria externa"* para *CodeSimulink*. Los nuevos módulos, de nombre `sim_extRAMread` y `sim_extRAMwrite`, ofrecen lectura y escritura en un chip de memoria *RAM* externo a la *FPGA*. El código *VHDL* escrito es genérico,[3] y por tanto, no depende de la placa aceleradora concreta ni de la *FPGA* que use sus recursos.
No obstante, se ha particularizado una copia de esos módulos para contener toda la información necesaria de patillaje, reloj etc, de la tarjeta *RC–1000* y se ha añadido a la nueva biblioteca *AlphaData* de *CodeSimulink*. En este contexto, se han creado los siguientes 8 nuevos bloques *CodeSimulink*: sim_extRAMread_RC1000_0 ... sim_extRAMread_RC1000_3 para la interfaz de lectura con los 4 módulos de memoria externa de la placa *RC1000*[4], y sim_extRAMwrite_RC1000_0 ... sim_extRAMwrite_RC1000_3 para la interfaz de escritura sobre

[3] Síntesis lógica comprobada para varios modelos de *FPGAs* de *Altera* y de *Xilinx*
[4] Nótese el sufijo *_RC1000* indicativo de que el módulo está particularizado para la tarjeta *RC1000* de *AlphaData*.

Parámetro	Valores	Valor por defecto
Longitud FIFO	natural_0	4
Valor del borde de máscara	real	128
Decimación de columnas	natural	1
Decimación de filas	natural	1
Valores de mascara	matriz real	ninguna
Implementación	*HSM_CSHI*	`parallelSe-rialMatrix`
Long. de datos internos	natural	16
Pos. punto flotante	entero	+8
Representación datos interna	*CS_SR*	Signed
Opción de desbordamiento interno	*CS_OV*	Saturation
Algoritmo de redondeo interno	*CS_RO*	Round
Long. de datos internos	natural	Saturation
Opción de desbordamiento	*CS_OV*	Saturation
Desbordamiento multiplicadores	*CS_OV*	Saturation
Redondeo multiplicadores	*CS_RO*	Round

Tabla 6.1: Parámetros del bloque `sim_convolver`

los módulos de la misma tarjeta. La figura 6.8 (página 97) muestra una captura de pantalla que visualiza la nueva biblioteca *AlphaDataRC1000* de *CodeSimulink*, que contiene los bloques de acceso a *RAM* definidos.

Los dos interfaces con memoria *RAM*, permiten definir una ventana de trabajo donde leer y escribir datos, que depende de una dirección base para el módulo de memoria en cuestión, que puede no coincidir con la dirección base física del módulo. Por tanto, el acceso a *RAM* se realiza según el esquema de la figura 6.9.

La tabla 6.2 muestra los parámetros *CodeSimulink* del bloque de acceso a *RAM* externa `sim_extRAMread`. Como es usual, estos parámetros son accesibles haciendo *doble click* sobre el bloque en cuestión. La figura 6.12 muestra una captura gráfica del cuadro de diálogo del bloque `sim_extRAMread`, que habilita la edición de sus parámetros.

Como ya se comentaba, la ventana de trabajo dentro del módulo de memoria viene especificada por la dirección de la esquina superior izquierda de la ventana de imagen (tomando como referencia la dirección base[5]), y por la anchura y altura de ésta (nº de columnas y filas de la imagen). También es posible decimar las direcciones de memoria por filas y por columnas de forma independiente. De esta forma si la decimación es 2 para las columnas, se direccionará una columna cada 2. Se deja al usuario el ajuste de las dimensiones de la ventana, de forma que cuadre con la decimación, ya que todos estos parámetros son libres. Como puede observarse en la figura 6.12, el bloque tiene cargados unos pines concretos, en este caso de la tarjeta *RC1000*, que están pre–asignados de forma oportuna. No obstante, podemos cargar la información de

[5]Nótese que el inicio de coordenadas para la memoria comienza en 0.

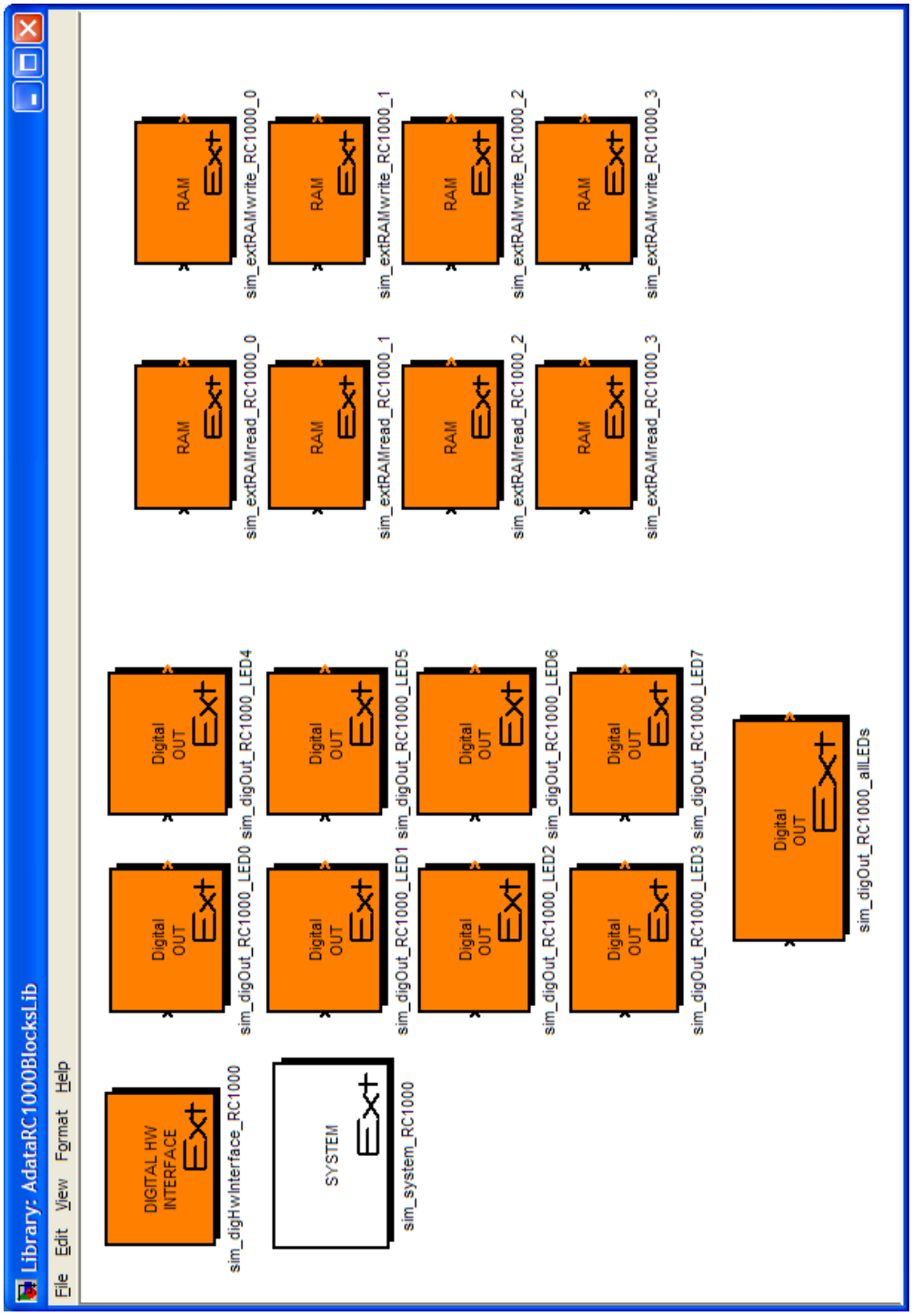

Figura 6.8: Nueva biblioteca *AlphaData* para *CodeSimulink*.

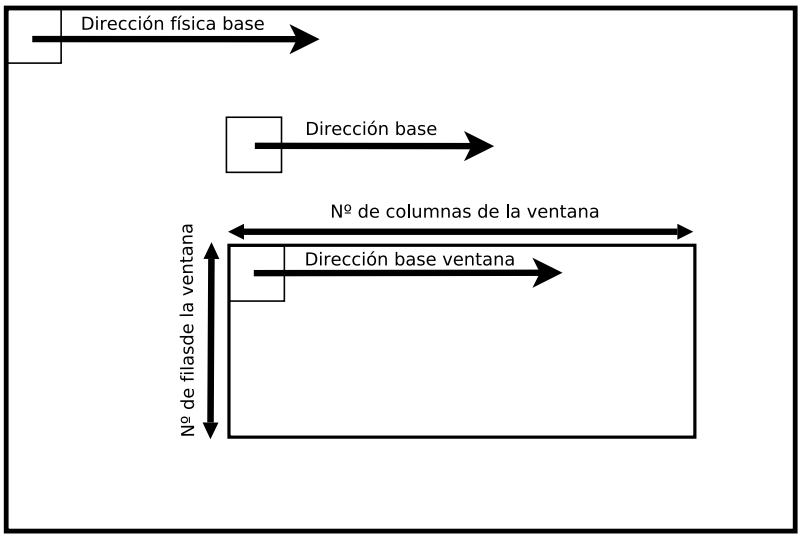

Figura 6.9: Definición en *RAM* externa de una ventana de trabajo.

Parámetro	Valores	Valor por defecto
Columnas matriz de vídeo	natural	160
Filas matriz de vídeo	natural	120
Primera columna	natural_0	0
Primera fila	natural_0	0
Nº columnas ventana	natural	160
Nº filas ventana	natural	120
Decimación de filas	natural	1
Decimación de columnas	natural	1
Longitud de palabra *RAM*	natural	32
Longitud de direccionamiento	natural	21
Dirección *RAM* base	natural_0	0
Estados de espera	natural_0	0

Tabla 6.2: Parámetros de `sim_extRAMread`

cualquier otra tarjeta soportada por *CodeSimulink* y tener así accesible los nuevos pines. También podemos directamente cargar la información de una *FPGA* individual (sin relación con ninguna tarjeta). El proceso de *CodeSimulink* para adaptar cualquier módulo de una tarjeta o *FPGA* es automático y toma pocos segundos.
Accediendo al botón *"Digital Hw Parameters"*, dentro del ámbito *Output Properties* (ver figura 6.12), se accede a la edición de los parámetros de salida del bloque, tal y como muestra la figura 6.11. Las dimensiones de la matriz de salida se calculan automáticamente de los datos del cuadro de diálogo anterior. En este nuevo cuadro, pueden editarse las características de la señal de salida

típicas de cualquier señal *CodeSimulink*. En el caso de la figura que nos ocupa, tenemos una señal de tipo matriz 120×160 de datos de 32 bits sin signo. El nivel de pipeline es 1. Dado que en este ejemplo concreto, los datos que provienen de la *RAM* tienen el formato *RGB* y cada canal cromático 8 bits, nos conviene que el *Manejo de datos no representables* sea de tipo *wraparound*, que infiere menos lógica. El método de redondeo escogido es *Round*.

El bloque *CodeSimulink* para escritura en *RAM* externa es el dual del bloque de lectura. Consta de los parámetros mostrados en la tabla 6.3.

Parámetro	Valores	Valor por defecto
Columnas matriz de vídeo	natural	160
Filas matriz de vídeo	natural	120
Primera columna	natural_0	0
Primera fila	natural_0	0
Factor de Zoom en filas	natural	1
Factor de Zoom en columnas	natural	1
Longitud de palabra *RAM*	natural	32
Longitud de direccionamiento	natural	21
Dirección *RAM* base	natural_0	0
Estados de espera	natural_0	0

Tabla 6.3: Parámetros de `sim_extRAMwrite`

Como puede comprobarse en dicha tabla, se debe especificar unas dimensiones para la matriz que se va a escribir en *RAM* y también las coordenadas de la ventana de escritura. La decimación pasa a llamarse *Zooming factor* aunque el parámetro mantiene el mismo comportamiento. Todas las demás características son similares al bloque de lectura, en particular el cuadro de diálogo de propiedades de la señal de salida, que es idéntico al mostrado en la figura 6.11.

Bloques de utilidad en sistemas de visión

Unos módulos interesantes para trabajar con visión en *CodeSimulink* son los módulos *sim_splitRGB* y *sim_unsplitRGB*, cuyos símbolos pueden verse en la figura 6.13. Generalmente en visión, se maneja el espacio de colores `RGB`. Estos nuevos bloques facilitan el manejo de estas señales. A cada uno de los bloques se le precisa el número de bits que tendrá cada uno de los 3 planos de color `R`, `G` y `B`. Aparte de separar una señal `RGB` en sus componentes o de concatenar 3 señales para conformar una nueva señal `RGB`, estos bloques permiten modificar las señales de salida que producen según *CodeSimulink*. Es decir, podemos modificar:

1. Longitud de los datos.
2. Posición del punto decimal.
3. Representación de la señal (signed, unsigned, ...).

4. Manejo de datos no representables (Saturation, Wraparound, ...).
5. Método de redondeo (Round, Floor, ...).

que hace que estos bloques no sean unos simples *splitters* o *unsplitters* de vectores de bits.

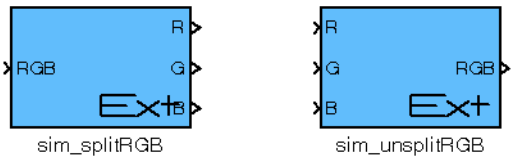

Figura 6.13: Bloques splitter diseñados para *CodeSimulink*.

6.3.4. Bloques software

El principal bloque software que se ha implementado es la interfaz con una webcam. Este bloque (cuyo símbolo muestra la figura 6.14), aunque no genera hardware, es de gran utilidad, ya que dota a *Simulink* de la capacidad de generar una entrada de vídeo a todos sus bloques y de forma particular a *CodeSimulink*. El comportamiento de sus parámetros es ligeramente distinto al de resto de los bloques *HSM*. Por defecto, si tenemos una videocámara compatible con la biblioteca *Vídeo for Windows* conectada al *PC*, entonces tomará de ella los parámetros oportunos para la anchura y altura de la imagen que genera. Si no tenemos ninguna cámara en a nuestro *PC*, entonces tomará por defecto las dimensiones 120×160 para la imagen, con 3 planos de color y 8 bits por plano de color para cada *pixel*.

Figura 6.14: Bloque software videocámara

6.4. Conclusiones

En este capítulo se ha presentado el *Módulo de modelado en hardware reconfigurable* de la plataforma de visión que nos ocupa.

Previamente se ha realizado un análisis de los requisitos hardware de los sistemas de visión que han influido a la hora de diseñar la plataforma y definir la arquitectura de referencia para el sistema de visión.

Posteriormente se ha discutido la elección del entorno de desarrollo *CodeSimulink–Matlab–Simulink* como la herramienta programática ideal para elaborar la plataforma. Finalmente se ha expuesto la propuesta en hardware reconfigurable para la realización del sistema de visión, y se han presentado los nuevos bloques *VHDL* compatibles con *CodeSimulink* que se han escrito para las tareas de procesamiento de imágenes. El módulo de cálculo principal, el *convolver*, admite tres posibles implementaciones con distinto grado de paralelismo (serie, paralelo y paralelo-serie,). Ajustando el tamaño y los coeficientes del módulo es posible aplicar cualquier máscara de convolución que aproxime el correspondiente filtro espacial definido en *Retiner*.
Por otra parte, los módulos de acceso a *RAM* externa definidos son esenciales para gestionar el trasiego de datos de la cámara de vídeo a la *FPGA*, y de ésta a la memoria externa de la interfaz con el *PC*.
El cómputo de módulos definidos constituye una biblioteca que amplia las posibilidades de *CodeSimulink* en aplicaciones que combinan el procesamiento de imágenes con otras formas de procesamiento de señales.

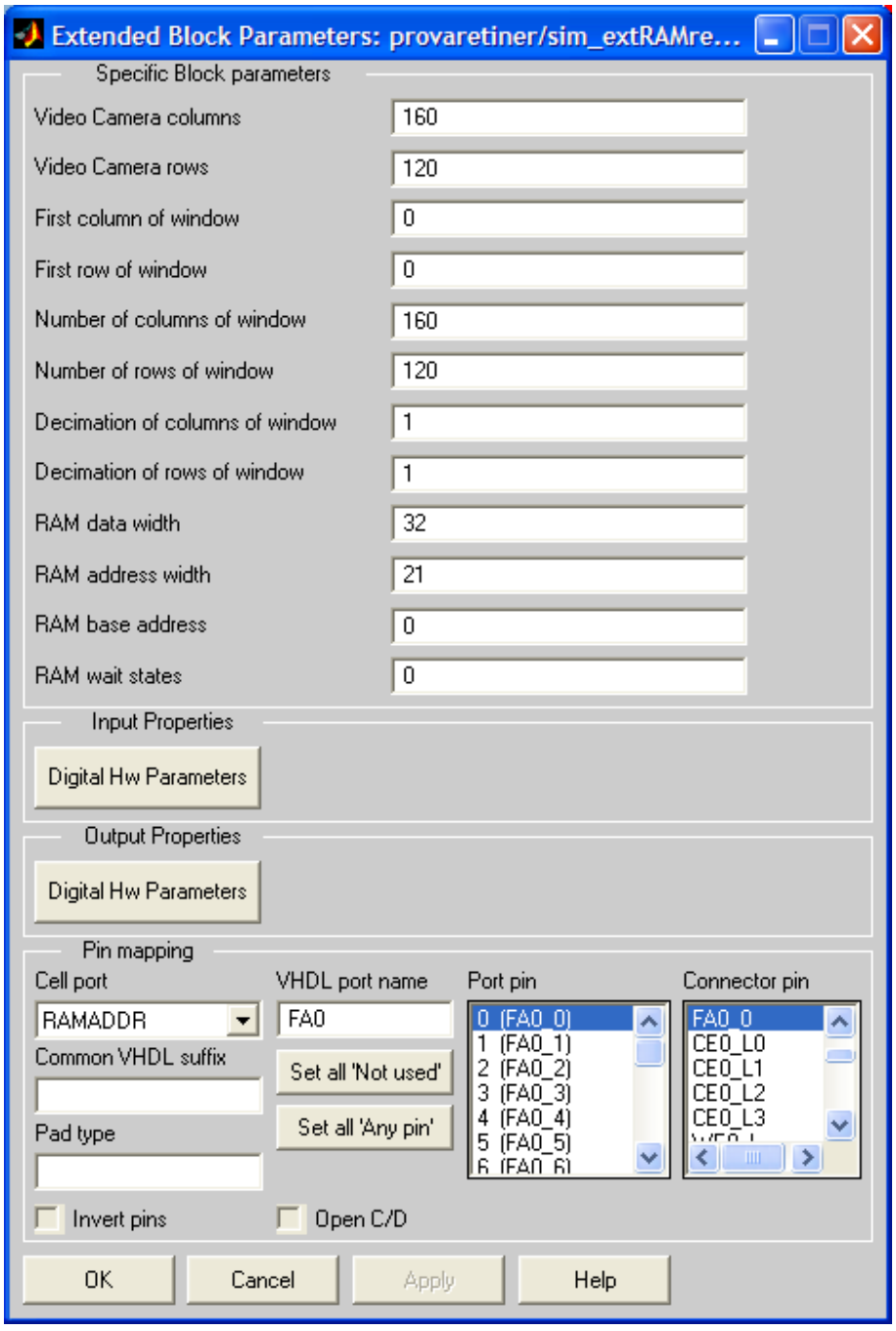

Figura 6.10: Parámetros del bloque `sim_extRAMread_RC1000`.

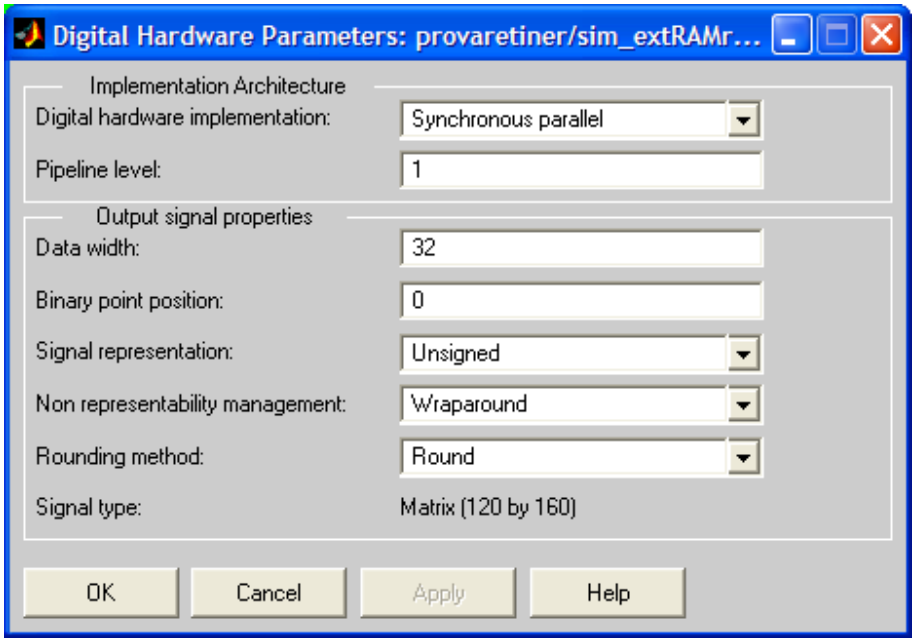

Figura 6.11: Parámetros de la señal de salida del bloque `sim_extRAMread_RC1000`.

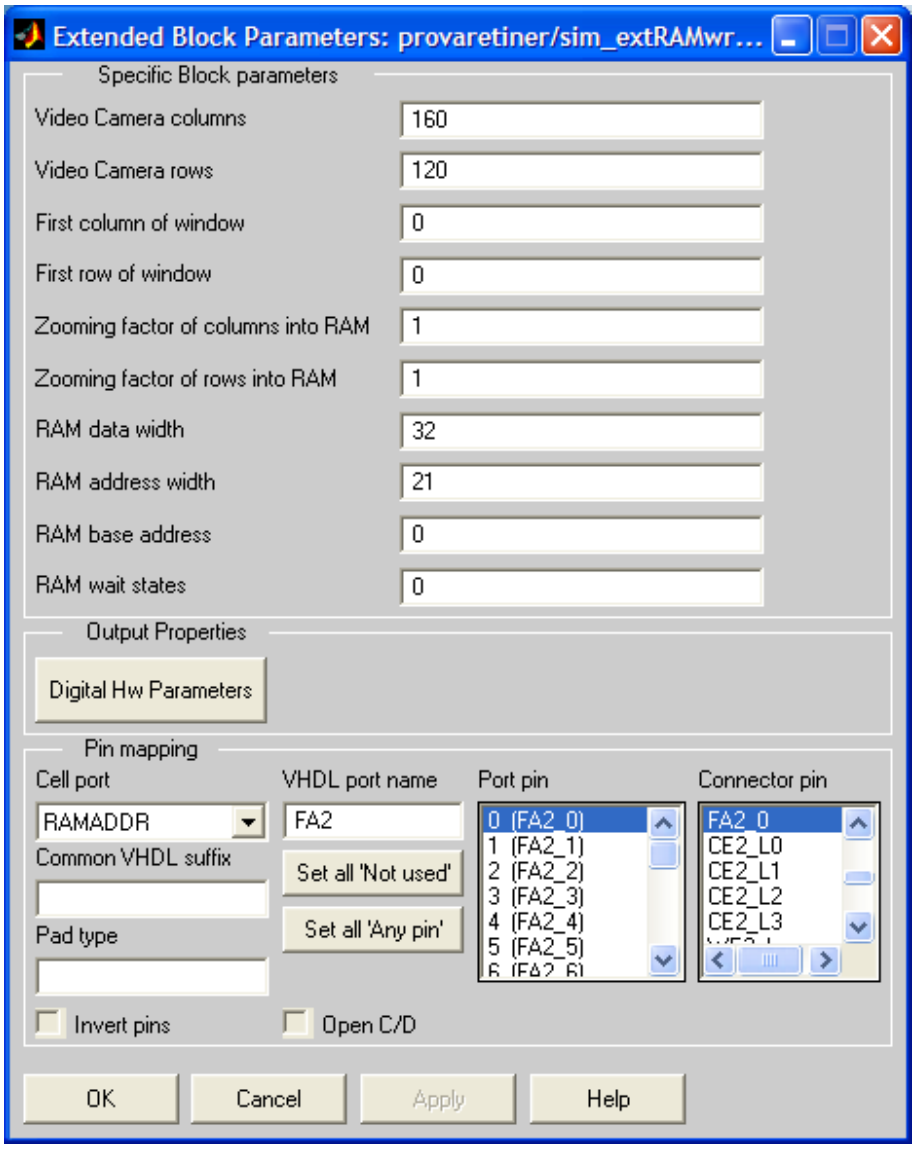

Figura 6.12: Parámetros del bloque `sim_extRAMread_RC1000`.

Capítulo 7

Generación automática del sistema de procesamiento. La herramienta *HSM*.

En el presente capítulo se presenta el Módulo de síntesis automática de alto nivel *definido en el capítulo 4. La sección 7.1 hace una introducción al capítulo y lo articula convenientemente. La sección 7.2 describe y analiza todo el proceso de translación a bloques hardware de un modelo funcional de visión, así como los distintos procesos de optimización que lleva a cabo* HSM.

7.1. Introducción

La generación automática Hw/Sw de sistemas de visión a partir de especificaciones generales de diferentes modelos, es una tarea que se nos presenta compleja y nos demanda una estructuración bien definida, tanto de las estrategias de generación automática de hardware y de software, como de las herramientas de especificación funcional de alto nivel que llevarán a cabo todo el recorrido descendente de síntesis y simulación del modelo.

En este sentido, el esfuerzo principal en el presente trabajo de tesis doctoral se ha focalizado en *Retiner*, descrito en la sección 5.2, como la herramienta principal de especificación y configuración de alto nivel de esquemas bio–inspirados de visión. No obstante, como veremos más adelante, el sistema completo propuesto, cuyo nombre es *HSM* (acrónimo de *Hardware Software Maker*), permite definir el modelo de visión a través de otros *puntos de entrada de especificación* a la cadena de generación automática, distintos del proporcionado por *Retiner*.

El presente capítulo describe las herramientas de la plataforma *HSM* mostrando sus capacidades, aplicaciones y características más relevantes.

Posteriormente,y suponiendo que parecerá al lector más ilustrativo, se describe la generación automática de hardware y de software a partir de una des-

cripción del sistema de visión especificada por *Retiner*, siendo éste un caso particular de las posibles entradas de especificación de sistemas de visión que podemos elegir con la plataforma *HSM*.

7.2. *HSM*: generación automática de hardware y software

La plataforma *HSM* contiene varios módulos y aplicaciones que llevan a cabo la traducción automática de una entrada de diseño de un modelo de visión concreto y en un formato específico, a modelos duales en *VHDL*. La estructura software de la herramienta está diseñada siguiendo un modelo de componentes que ayude a una mejor depuración y organización de las distintas tareas a desarrollar. Esta estrategia de diseño hace más fácil y efectivo modificar y añadir nuevas características a las aplicaciones.

Como se comenta en el capítulo 4, la plataforma *HSM* es un conjunto de bibliotecas, aplicaciones y módulos *VHDL* que conforman juntos una plataforma de desarrollo de sistemas de visión. A diferencia del capítulo 6, en el que se han descrito los módulos y el modelo hardware que utiliza *HSM*, en el presente capítulo se describen las aplicaciones software de *HSM* que llevan a cabo la traducción antes mencionada.

7.2.1. Generación a partir de *Retiner*

El objetivo principal de la herramienta *Retiner* es el de proporcionar un sistema de diseño, evaluación y validación de esquemas de visión por ordenador que está, como hemos comentado, especialmente orientada a la conformación de retinas artificiales. Este sistema, es un ejemplo en sí mismo de especificación funcional de un sistema complejo en muy alto nivel. El modelo de visión de *Retiner*, es de tipo matemático y queda definido por un conjunto conocido de filtros de visión y una relación matemática entre ellos (consultar los capítulos 3 y 5).
Para dotar a *Retiner* de la capacidad de generar descripciones sintetizables a partir de sus modelos de visión, se ha diseñado un *plugin* o extensión a esta herramienta que desempeña esta tarea. Esta extensión toma el nombre de *HSM–RI*.

Especificación del modelo sintetizable: La herramienta *HSM–RI*

HSM–RI, acrónimo de *HSM – Retiner Interface*, hace las labores de interfaz entre *Retiner* y la herramienta *HSM*.

La figura 7.1 representa el esquema del proceso de síntesis que sigue *HSM* cuando la entrada al flujo de diseño se realiza por medio de la herramienta *Retiner*.

Distingo varias etapas dentro de este flujo de diseño:

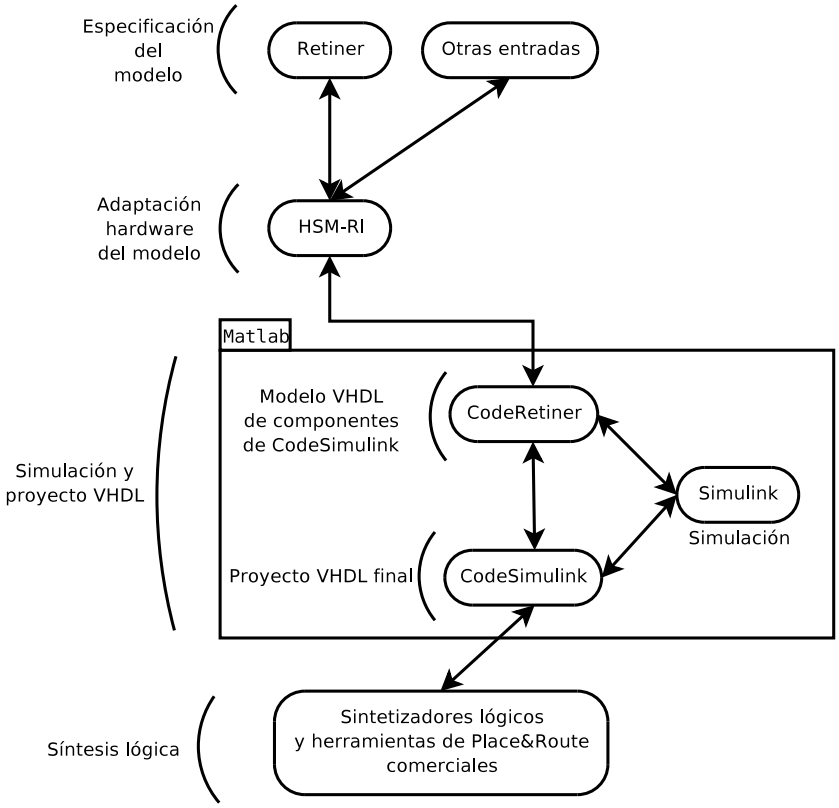

Figura 7.1: Esquema del proceso de síntesis llevado a cabo por *HSM*.

- **Especificación del modelo**

  Descrito mediante la herramienta *Retiner*, la entrada al flujo de diseño es una descripción funcional de un sistema de visión expresada con una ecuación matemática sencilla. El modelo de visión viene por tanto determinado por un conjunto definido de filtros y una relación matemática entre ellos. Este modelo se puede simular funcionalmente con *Retiner* hasta validarlo.

- **Adaptación Hardware del modelo**

  Donde viene adaptado el modelo de alto nivel de especificación proporcionado por *Retiner* a una descripción susceptible de ser trasladable a hardware. La principal aplicación en esta tarea es *HSM–RI*, que transforma la expresión de partida a un modelo de componentes hardware conformado por primitivas conocidas de una librería de componentes soportada por *HSM*. El resultado final de esta etapa es un nuevo modelo análogo al

modelo de partida de *Retiner* salvo una serie de restricciones y optimizaciones que luego detallaré. En adelante me referiré a este modelo como modelo optimizado.

- **Simulación y proyecto *VHDL***

  En esta etapa el nuevo modelo viene nuevamente descrito mediante las primitivas hardware de la biblioteca de componentes *CodeSimulink*. Una vez realizada dicha tarea, el modelo se proyecta a su descripción gemela en *Simulink* para su posterior simulación y síntesis. También en este nivel, la aplicación *CodeSimulink* es capaz de elaborar un proyecto VHDL listo para ser procesado por sintetizadores lógicos comerciales. Se ha denominado *CodeRetiner* a la herramienta que genera el modelo *CodeSimulink* a partir del modelo optimizado que ofrece *HSM–RI*.

- **Síntesis lógica**

  En esta etapa, sintetizadores comerciales como *Leonardo Spectrum* ®, *Xilinx ISE* ®, *Altera Max+Plus II* ® etc. se encargán de transladar a la tecnología escogida (*FPGA* o *ASIC*) el proyecto *VHDL* generado en la etapa anterior.

Adaptación Hardware del modelo

La descripción del sistema de procesamiento visual viene dada mediante *Retiner*, aplicación que trabaja en la capa de especificación del modelo funcional (ver figura 7.1). Esta descripción está sujeta a las restricciones propias de dicha aplicación (consultar la sección 5.2) más las que son impuestas por la nueva herramienta *HSM–RI* (*HSM–Retiner Interface*), que comienza el proceso de traducción de la expresión. Es decir, sólo podemos sintetizar en hardware un subconjunto de las descripciones de modelos de visión que podemos simular con *Retiner*. Al no tratarse esta expresión de una cualquiera de las permitidas por *Retiner* (que permite incluir cualquier función *Matlab*), en adelante me referiré a ella como *RES* (*Expresión Retiner para Síntesis*) en contraposición a un expresión *Retiner* general (*RE*). Destaco por claridad la definición de estas dos expresiones:

RE Del inglés *Retiner Expression*. Es una descripción matemática de un sistema de visión por ordenador que soporta la aplicación *Retiner* (consultar la sección 5.2).

RES Del inglés *Retiner Expression for Synthesis*. Es una *RE* susceptible de ser transladable a hardware por la herramienta *HSM–RI*. Las expresiones *RES* son un subconjunto de la *RE*.

La tabla 7.1 muestra las funciones y operadores que pueden incluirse dentro de una expresión *RES* en la versión actual de *HSM*.

De este modo, a partir de una descripción de una retina definida en *Retiner* de forma matemática, *HSM* comienza el proceso de traducción del sistema por

Tabla 7.1: Expresiones Matlab–*Retiner* permitidas por *HSM–RI*

Nombre	**Parámetros**	**Significado**	**func. *Matlab***	**func. *Retiner***
`DoG`	`RP`, σ_1, σ_2, L, I_1, I_2	Diferencia de Gaussianas		•
`Gauss`	`RP`, σ, I, L	Gaussianas		•
$+\ -\ \times\ /\ (\ )$		Expresión algebráica	•	•
`LoG`	`RP`, σ_1, σ_2, L, I	Laplaciana de Gaussianas		•
`R`, `G`, `B`, `I`		Canáles cromáticos y canal acromático		•

Donde los parámetros σ denotan la desviación típica de las gaussianas que intervienen en el filtrado *DoG* (diferencia de dos gaussianas) y *Gauss*. L en todos los casos denota la longitud de la máscara o kernel de convolución, que es una matriz de dimensión $L \times L$. I_i (la entrada principal de la función) es una combinación lineal de primitivas anteriores y de los canales cromáticos y acromáticos (R, G, B e I). El parámetro RP (*Retiner Parameters*) sólo puede valer [0] o [1] y es una constante de tipo matriz 1×1 en *Matlab* que puede tomar distintos significados con distintas funciones de *Retiner*, en la presente versión (1.0) y para todas las funciones, 0 significa que no se normaliza la salida del filtro en cuestión y 1 lo contrario. Las dos últimas columnas indican si se trata de una expresión o función de *Matlab* o de *Retiner*.

medio de la herramienta HSM–RI. Como se ha comentado, de todas las posibles funciones que podemos integrar dentro de una expresión *RE*, sólo una parte de ellas podrá dar lugar a una implementación en hardware (ver tabla 7.1).

Una descripción general de un sistema de visión podría tener la forma de la ecuación 7.4; en donde los términos f_i se corresponden con las distintas funciones de filtro (*DoG*, *Gauss*, *LoG*...).

$$\begin{aligned} f_1 &= DoG(RP_1, \sigma_{11}, \sigma_{12}, L_1, I_{11}, I_{12}; \ldots) \\ f_2 &= DoG(RP_2, \sigma_{21}, \sigma_{22}, L_2, I_{21}, G_{22}; \ldots) \\ &\vdots \\ f_N &= DoG(RP_3, \sigma_{N1}, \sigma_{N2}, L_N, I_{N1}, I_{N2}; \ldots) \end{aligned} \tag{7.1}$$

$$\begin{aligned} f_{N+1} &= Gauss(RP_{N+1}, \sigma_{N+1}, L_{N+1}, I_{N+1\,1}; \ldots) \\ f_{N+2} &= Gauss(RP_{N+2}, \sigma_{N+2}, L_{N+2}, I_{N+2\,2}; \ldots) \\ &\vdots \\ f_{N+M} &= Gauss(RP_{N+M}, \sigma_{N+M}, L_{N+M}, I_{N+M\,M}; \ldots) \end{aligned} \tag{7.2}$$

$$\begin{aligned} f_{N+M+1} &= LoG(RP_{N+M+1}, \sigma_{N+M+1}, L_{N+M+1}, I_{N+M+1}; \ldots) \\ f_{N+M+2} &= LoG(RP_{N+M+2}, \sigma_{N+M+2}, L_{N+M+2}, I_{N+M+2}; \ldots) \\ &\vdots \\ f_{N+M+L} &= LoG(RP_{N+M+L}, \sigma_{N+M+L}, L_{N+M+L}, I_{N+M+L}; \ldots) \end{aligned} \tag{7.3}$$

$$Sistema = a \times f_1 + b \times f_2 + c \times f_3 + \ldots \tag{7.4}$$

Veamos a continuación algunos ejemplos concretos de descripciones *Retiner* de posibles sistemas de procesamiento visual. La ecuación 7.5 muestra la expresión matemática de un sistema de visión que aproxima un filtrado espacial inspirado en el de una retina humana. En ella nos encontramos la actuación de una suma de dos diferencias de gaussianas (*DoG*) y un laplaciano de gaussianas (*LoG*) que procesan diferentes combinaciones lineales de entrada de los canales cromáticos (R, G y B) y el acromático (I). Las dos primeras *DoG* tienen el valor 1.2 para el parámetro σ_1 y 0.9 para el σ_2 (desviaciones típicas). Todos los filtros son bidimensionales y espaciales, es decir, no interviene la variable tiempo y por tanto cada frame calculado no depende del anterior:

$$\begin{aligned} Sistema_1 = \frac{1}{6} \cdot (&2 \cdot DoG([1], 1.2, 0.9, 7, R+B, G) \\ &+3 \cdot DoG([1], 1.2, 0.9, 7, R+G, B) + LoG([1], 1.4, 7, I)) \end{aligned} \tag{7.5}$$

Esta expresión puede ser procesada directamente por *HSM–RI* que, después de traducirla y generar una descripción apropiada en bloques hardware

de comportamiento simétrico, podrá ser transladada correctamente a una tecnología de circuito integrado, concretamente a una *FPGA* o celdas estándar.

HSM–RI realiza una serie de minimizaciones de la expresión de forma que, el resultado sea menos costoso en hardware o venga calculado más rápido.

HSM–RI lleva a cabo una serie etapas en el procesamiento de una *RES*[1], que se muestran a continuación:

1. Análisis sintáctico de la expresión (*parsing*).
   - Evaluación de la sintaxis de la expresión matemática.
   - Evaluación de la sintaxis de cada función empleada.
2. Extracción de señales
   - Minimización algebráica de señales.
   - Supresión de señales innecesarias.
3. Expansión de la expresión matemática.
4. Reorganización de las primitivas de cálculo y las señales
   - Inferencia de funciones menos costosas computacionalmente.
   - Reescritura de la expresión en función de primitivas hardware conocidas.

Para ilustrar más gráficamente algunas de las minimizaciones que proporciona la herramienta, suponiendo que la entrada desde *Retiner* es la mostrada en la ecuación 7.6:

$$Sistema_2 = \frac{3}{2}(4\,DoG([1], \sigma_1, \sigma_2, l, 2\,R+G, B) + 2\,DoG([1], \sigma_1, \sigma_2, 2\,R+G, 2\,B)) \tag{7.6}$$

En este ejemplo, cada filtro *DoG* será implementado mediante sendas convoluciones gaussianas sobre el minuendo y el sustraendo de la diferencia de gaussianas. Cuando estos dos son iguales y sólo varía el parámetro σ (la desviación típica), sólo es necesario calcular una convolución, con máscara de convolución la resta de las máscaras gaussianas. A este tipo de filtro se le conoce como *sombrero mexicano*, por su parecido con dicha prenda. *HSM–RI* es capaz de detectar que en algunos casos sólo es necesario el empleo de una sóla convolución. No obstante, lo habitual cuando tratamos de modelar una retina humana es que tanto minuendo como sustraendo de la diferencia de gaussianas

[1] *RES*: Expresión *Retiner* para síntesis

sean distintos, como ya vimos en el ejemplo 7.5; así que en la mayoría de los casos más interesantes que nos ocupan, tal optimización no será posible. Esta situación es usual cuando se realizan filtrados cromáticos, en los que estamos especialmente interesados en la información extraída de cada uno de los canales de color.

Otra minimización importante que puede llevar a cabo *HSM–RI* cuando se encuentra con varias funciones que implican una convolución (como la *DoG*), es la de operar en el espacio de operadores aplicando las propiedades del operador convolución para reducir el coste computacional de la expresión original. En este sentido, consideremos el ejemplo de sistema representado en la ecuación 7.7.

$$\begin{aligned} Sistema \quad = \quad & \frac{1}{4}(2 \cdot DoG([RP], \sigma_1, \sigma_2, L, R+B, G) + \\ & DoG([RP], \sigma_3, \sigma_4, L, R+G, B) + \\ & DoG([RP], \sigma_5, \sigma_6, L, \frac{R+G+B}{3}, \frac{R+G+B}{3})) \end{aligned} \tag{7.7}$$

Internamente, desarrollando las diferencias de gaussianas, tendríamos en principio que calcular 6 convoluciones distintas comandadas por los distintos parámetros $\sigma_1 \ldots \sigma_6$ que aparecen. Al contrario que el anterior ejemplo, ahora no podemos operar entre las gaussianas de las distintas *DoG* porque no tenemos núcleos iguales, no obstante, sí que podemos operar si representamos la expresión en el espacio de operadores. De este modo, la ecuación 7.7 puede ser re–escrita para dar la nueva ecuación 7.8.

$$\begin{aligned} Sistema \quad = \quad & \frac{1}{2}(\Gamma_1 * R + \Gamma_1 * B - \Gamma_2 * G) + \\ & \frac{1}{4}(\Gamma_3 * R + \Gamma_3 * G - \Gamma_4 * B) + \\ & \frac{1}{4}(\Gamma_5 * \frac{R}{3} + \Gamma_5 * \frac{G}{3} + \Gamma_5 * \frac{B}{3} - \\ & \Gamma_6 * \frac{R}{3} - \Gamma_6 * \frac{G}{3} - \Gamma_6 * \frac{B}{3}) \end{aligned} \tag{7.8}$$

en donde:

$$\Gamma_i = Gauss(\sigma_i, \cdot) \tag{7.9}$$

con $i \in \{1 \ldots 6\}$ las diferentes gaussianas que conforman las 3 funciones *DoG* que se tienen en el ejemplo.

Una vez realizada este descomposición y reescritura en el espacio de operadores, *HSM* es capaz de minizar esta expresión para dar:

$$Sistema = \kappa_R \star R + \kappa_G \star G + \kappa_B \star B \tag{7.10}$$

con

$$\begin{aligned} \kappa_R &= \frac{1}{2}\cdot\Gamma_1 + \frac{1}{4}\cdot\Gamma_3 + \frac{1}{12}\cdot\Gamma_5 - \frac{1}{12}\cdot\Gamma_6 \\ \kappa_G &= -\frac{1}{2}\cdot\Gamma_2 + \frac{1}{4}\cdot\Gamma_3 + \frac{1}{12}\cdot\Gamma_5 - \frac{1}{12}\cdot\Gamma_6 \\ \kappa_B &= \frac{1}{2}\cdot\Gamma_1 - \frac{1}{4}\cdot\Gamma_4 + \frac{1}{12}\cdot\Gamma_5 - \frac{1}{12}\cdot\Gamma_6 \end{aligned} \tag{7.11}$$

donde se observa que finalmente se ha reducido a la mitad el número de convoluciones deben realizarse.

Todas las optimizaciones que lleva a cabo *HSM* son parametrizables y opcionales. Además, puede variarse el orden de ejecución de las mismas automáticamente por parte del mismo *HSM* o por parte del usuario para intentar conseguir resultados más óptimos.

Lo primero que hace *HSM–RI* con la expresión 7.6 es analizar su sintaxis, de forma que verifique la estructura algebraica de una combinación lineal de variables y funciones. Una vez realizada esta operación, se pasa a generar una lista con todas las señales que intervienen en la expresión. Defino a continuación algunos conceptos de forma estricta.

Canal Cada una de las distintas componentes que forman un *pixel* o que dan información relevante sobre él en la imagen. Generalmente, un sistema de visión que procese información dentro del espectro visible, suele estar alimentado de las tres componentes o canales cromáticos R, G y B (respectivamente la componente roja, verde y azul que conforma un pixel). *HSM–RI* supone siempre que existen estas tres variables más el canal acromático o de intensidad I. A estas 4 variables les llamaré canales primitivos.

Señal Toda combinación lineal de los canales primitivos y las funciones *RES*

Tanto *Retiner* como *HSM–RI* suponen 4 canales primitivos que no necesitan ser definidos, R, G, B e I. El canal I se calcula por defecto como la media aritmética de los canales cromáticos R, G y B. No obstante, todos los canales pueden ser redefinidos dentro de *HSM–RI* de forma que pueden valer cualquier combinación lineal entre todos los canales que se tiene disponibles siempre y cuando no haya referencias cruzadas entre ellos o que un canal dependa de sí mismo.

Simulación y proyecto *VHDL*

Una vez que *HSM–RI* ha obtenido una expresión minimizada a partir del modelo *RES* de entrada, entra en juego el siguiente módulo que refleja la figura 7.1 y cuyo nombre es *CodeRetiner*. Esta utilidad está disponible tanto en un programa individual en línea de órdenes como en distintas funciones del *API* de

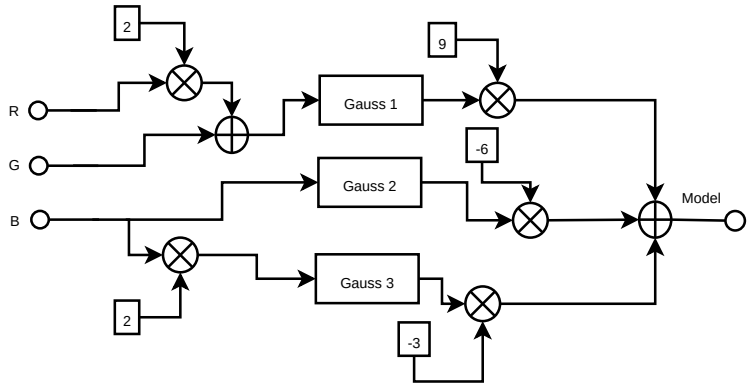

Figura 7.2: Esquema de la estructura inferida

HSM. Su nombre da cuenta de que es la utilidad que se encarga directamente de realizar la interfaz entre *Retiner* y la biblioteca de funciones de *CodeSimulink*.

CodeRetiner, se encarga de generar automáticamente una función en *Matlab* partiendo de la descripción del sistema que le proporciona *HSM–RI*. Esta función se encarga de:

- Construir un modelo de bloques *CodeSimulink* a partir del modelo óptimo elaborado por *HSM–RI*. Se pueden crear dos tipos de modelos:

  Para simulación de alta precisión. Entendiendo alta precisión la más alta precisión que pueda soportar *Matlab*, que típicamente es de número real de 64 bits. Es decir en máquinas que soporten el estándar *IEEE* de aritmética para punto flotante de 64 bits [44], la precisión es de $2.2204e-16$, que es el mayor espacio relativo a dos números adyacentes en la representación de la máquina dada.

  Para simulación orientada a síntesis. En donde los bloques hardware inferidos tienen en cuenta las restricciones y la precisión propias del *hardware*. De las distintas posibilidades que ofrece *CodeSimulink* para caracterizar las señales (que pueden consultarse en el capítulo 3), *HSM–RI* puede manejar las siguientes:

  - Tipos de datos de entrada y salida (enteros, reales, aritmética de punto flotante o de punto fijo, etc . . .).
  - Longitud en número de bits de los datos.
  - Existencia o no de desbordamientos o saturación en los datos.
  - Tipo de representación de los datos (con signo, sin signo, módulo y signo, etc . . .).

- Relizar una simulación tanto del modelo en alta precisión, como del modelo orientado a síntesis.
- Llevar a cabo la compilación del diseño mediante las herramientas de *CodeSimulink* para generar un fichero *VHDL*, así como un fichero de proyecto del motor comercial de síntesis que se haya elegido (*Xilinx XST*, *Altera Quartus*, *Leonardo Spectrum*, etc ...).
- Llamar al motor de síntesis escogido para que realice la síntesis del proyecto, generando un fichero *EDIF*.
- Llamar a la herramienta de generación de ficheros de configuración elegida, para generar el archivo final del diseño, listo para ser transladado a la *FPGA* que convenga.
- Configurar la *FPGA* y, en los casos que sea posible, ejecutar un programa huésped de monitorización de la *FPGA* para comprobar el resultado final del diseño. En este paso también se permite la ejecución de un script opcional que debe suministrar el usuario.

En el caso de la simulación orientada a síntesis, se tiene la posibilidad de calcular automáticamente varios parámetros que inciden de forma decisiva en el tamaño y velocidad del circuito final. Activando esta opción, se calcula el rango dinámico de cada bloque *CodeSimulink* por separado, tanto para el modelo de alta precisión como para los orientados a síntesis. La presente versión de *HSM* es capaz de calcular automáticamente la longitud de palabra y la posición del punto flotante, para cada uno de los bloques, de:

- Todas las señales.
- Todos los coeficientes constantes utilizados, en especial los empleados internamente en los módulos *sim_convolver_syncpar* que se utilicen.

Añadir nuevos parámetros a optimizar es muy sencillo mediante la edición del código generado de la función ge A partir del calculo del rango dinámico de las distintas simulaciones, se calculan las *superficies de Pareto* de modelo para los distintos parámetros que se intentan optimizar.

También nuevamente, de forma opcional, se puede presentar la gráfica de dichas superficies y dejar la elección definitiva de los parámetros óptimos al usuario; en caso contrario *CodeRetiner* tomará automáticamente los puntos que se encuentren más cerca del origen y que tienen la cualidad de presentar la mejor relación en el compromiso entre superficie utilizada del chip y el valor del parámetro en cada caso.

Todos los parámetros de la función *CodeRetiner* están controlados por *Retiner*, no obstante también puede invocarse desde la línea de órdenes de *Matlab*).

Desde *Retiner* el usuario puede elegir quedarse en cualquiera de los pasos intermedios antes de sintetizar y comprobar todo el sistema.
Concretamente, es muy interesante la idea de crear un modelo hardware sintetizable en *CodeSimulink*, automáticamente a partir de *Retiner* y comparar los

resultados de la simulación del modelo, con los resultados que nos da *Retiner* del mismo.

Simulación y proyecto *VHDL*

Desde la especificación del modelo de visión por *Retiner*, podemos simular el sistema de las siguientes formas:

1. Utilizando *Retiner*. Para obtener una simulación de alta precisión como se indicaba en el apartado anterior.
2. Utilizando *CodeSimulink* para realizar otra simulación de alto nivel del modelo adaptado a bloques *CodeSimulink*.
3. Utilizando *CodeSimulink* para llevar a cabo una simulación compatible con las restricciones hardware de cada bloque y que se adapte al comportamiento hardware del sistema completo.
4. Utilizando un simulador comercial de *VHDL* como *ModelSim* o el motor de simulación de *Altera Max+PlusII* para simular el proyecto *VHDL* infiere *CodeSimulink*.

Capítulo 8

Resultados y validación experimental

El presenta capítulo presenta los resultados y validaciones experimentales más destacados del presente trabajo doctoral. Así, la sección 8.1 introduce brevemente la lista de experimentos que se han considerado. La sección 8.2 presenta los resultados de simulación funcional y síntesis, de un modelo concreto de visión, mostrando el funcionamiento de las distintas optimizaciones que se llevan a cabo en la herramienta. Posteriormente la sección 8.3 muestra la versatibilidad de Retiner *para simular y procesar los registros de impulsos que se obtienen al aplicar el modelo completo de retina + la generación de* spikes *y compararlos con los registros duales biológicos.*

8.1. Introducción

Este capítulo presenta los resultados de simulación funcional y síntesis de distintos esquemas de procesamiento visual, obtenidos mediante las herramientas desarrolladas durante el trabajo de tesis.

En esta línea, la sección 8.2 presenta los resultados obtenidos tanto en simulación funcional con *Retiner* como en síntesis con *HSM* y *CodeSimulink* de un cierto modelo de retina tomado como referencia. Se muestra la simulación de un modelo de referenca con *Retiner*, se translada dicho modelo a bloques *CodeSimulink*, y se presentan los resultados de aplicar varias optimizaciones, así como una comparativa de síntesis del mismo modelo realizada con *Handel-C*. Seguidamente la sección 8.3 presenta los resultados de contrastar datos biológicos y sintéticos tomando el mismo modelo de referencia.

8.2. Generación automática de un modelo de retina.

En esta sección se muestra la versatibilidad de la plataforma *HSM* para llevar a cabo la simulación y síntesis de un modelo concreto de retina, descrito

matemáticamente mediante la ecuación 8.1:

$$Modelo = \frac{1}{4} \cdot [2\, DoG(0.9, 1.2, 3, R+B, G) + DoG(0.9, 1.2, 3, R+G, B) + DoG(0.9, 1.2, 3, I, I)] \quad (8.1)$$

Como puede observarse, se trata de un caso particular de la ecuación 3.6[1] que modela el comportamiento retiniano. Se tiene una combinación lineal de tres filtros *DoG* con idénticos núcleos de convolución (dimensión 3×3, parámetro $\sigma_1 = 0.9$ y parámetro $\sigma_2 = 1.2$) que se aplican sobre distintas combinaciones de los canales cromáticos.

La combinación de los canales cromáticos cuya entrada alimenta a los filtros *DoG*, así como la ponderación de cada uno de estos filtros está basada en la biología de la retina humana de tal suerte que podemos establecer la equivalencia dada por la tabla 8.1:

$$DoG(0.9, 1.2, 3, R+B, G) \equiv DoG_{RB,G} \mapsto \text{o canal rojo-verde} \quad (8.2)$$
$$DoG(0.9, 1.2, 3, R+G, B) \equiv DoG_{RG,B} \mapsto \text{o canal amarillo-azul} \quad (8.3)$$
$$DoG(0.9, 1.2, 3, I, I) \equiv DoG_{I} \mapsto \text{o canal acromático} \quad (8.4)$$

Tabla 8.1: Equivalencia de los filtros utilizados en el modelo de referencia y el filtro de los canales cromáticos en el ser humano.

donde $DoG_{RB,G}$ modela el triplete retiniano fotorreceptores-bipolares-horizontales que realiza un realce espacial rojo + azul (o rojo-magenta) en centro frente a verde en periferia, $DoG_{RG,B}$ hace lo propio con el canal amarillo frente al azul y DoG_I lleva a cabo un realce espacial del canal acromático parecido a un detector de bordes sobre la imagen en escala de grises.

8.2.1. Simulación del modelo mediante *Retiner*.

El siguiente paso, una vez que se ha decidido el modelo matemático a emplear, es el de simular el sistema para comprobar el modelo y en su caso, adecuar los parámetros que convengan. Para ello, siguiendo un proceso similar al resumido en el Apéndice C, se especifica en *Retiner* el sistema de visión. De esta manera, una vez definidos los filtros y la combinación adecuada[2], *Retiner* los muestra en su ventana principal nombrándolos con la notación indicada por la tabla 8.2:

La asignación de nombres a los distintos filtros `F1, F2...` la proporciona *Retiner* de forma automática, una vez que ha procesado la dependencia de unos filtros con otros. Esto evita la necesidad de comprobar manualmente si la salida de un filtro forma parte de la entrada de otro.

[1] En el capítulo 3, página 53.

[2] Recuerde que el primer parámetro ([1] en este caso) significa que la salida del filtro estará normalizada en el intervalo $0 \ldots 255$.

```
      R
      G
      B
      I
     F1       =  f_DoG ([1], 0.9, 1.2, 3, R+B,G)
     F2       =  f_DoG ([1], 0.9, 1.2, 3, R+G,B)
     F3       =  f_DoG ([1], 0.9, 1.2, 3, I,I)

Combination   =  1/4 * (2*F1 + F2 + F3)
```

Tabla 8.2: Lista de canales y filtros definidos en *Retiner*, así como como la combinación escogida para modelar el comportamiento retiniano del ejemplo.

Una vez en este punto, y tal como se muestra a continuación, se ha procedido a la simulación en alto nivel mediante *Retiner* del modelo de visión que nos ocupa. En este caso, como se puede observar en la figura 8.1, se ha tomado una imagen estática (imagen superior) para probar de forma separada las respuestas de los distintos filtros $DoG_{RB,G}$, $DoG_{RG,B}$ y $DoG_{I,I}$, así como la combinación dada entre ellos. Se han añadido para cada simulación, el resultado del "*pixelado*"[3], que constituye una representación visual de la matriz de actividad, asi como un filtrado gaussiano adicional que produce un resultado más próximo a la imagen real evocada mediante un implante visual. Se han utilizado unas dimensiones para la matriz de microelectrodos de 10×10 y 25×25 respectivamente. Nótese que aun empleando las dimensiones 10×10 (dimensiones de la matriz de microelectrodos de la Universidad de *Utah* utilizada en *CORTIVIS*) para el *pixelado*, puede identificarse fácilmente el contorno, la cara y los ojos. Nótese también la diferente salida de los filtros y el peso del procesamiento cromático que se lleva a cabo en cada uno de ellos. En todos los casos, se han utilizado máscaras de convolución de dimensiones 3×3.

Con objeto de mostrar la diferencia en la aplicación de máscaras de convolución de distintas dimensiones, se adjunta la figura 8.2, en la que se muestra (al igual que la figura 8.1), el resultado de la combinación de la ecuación 8.1 con $\sigma_1 = 1.58$ y $\sigma_2 = 1.59$, junto con su salida *pixelada* y el filtrado gaussiano adicional. En este caso, las dimensiones de las máscaras o *kernels* de convolución son de 3×3 y 7×7 respectivamente, mientras que la matriz de microelectrodos es de 25×25. Mediante una sencilla inspección del resultado, se puede observar cómo la máscara de mayores dimensiones produce un suavizado más acentuado de la imagen, eliminando parte del ruido y evidenciando de forma más clara la actuación del filtro paso–baja que constituye la gaussiana.

[3]Resultado de asignar a cada electrodo un área determinada de la matriz de actividad, que pueden ser solapantes entre sí.

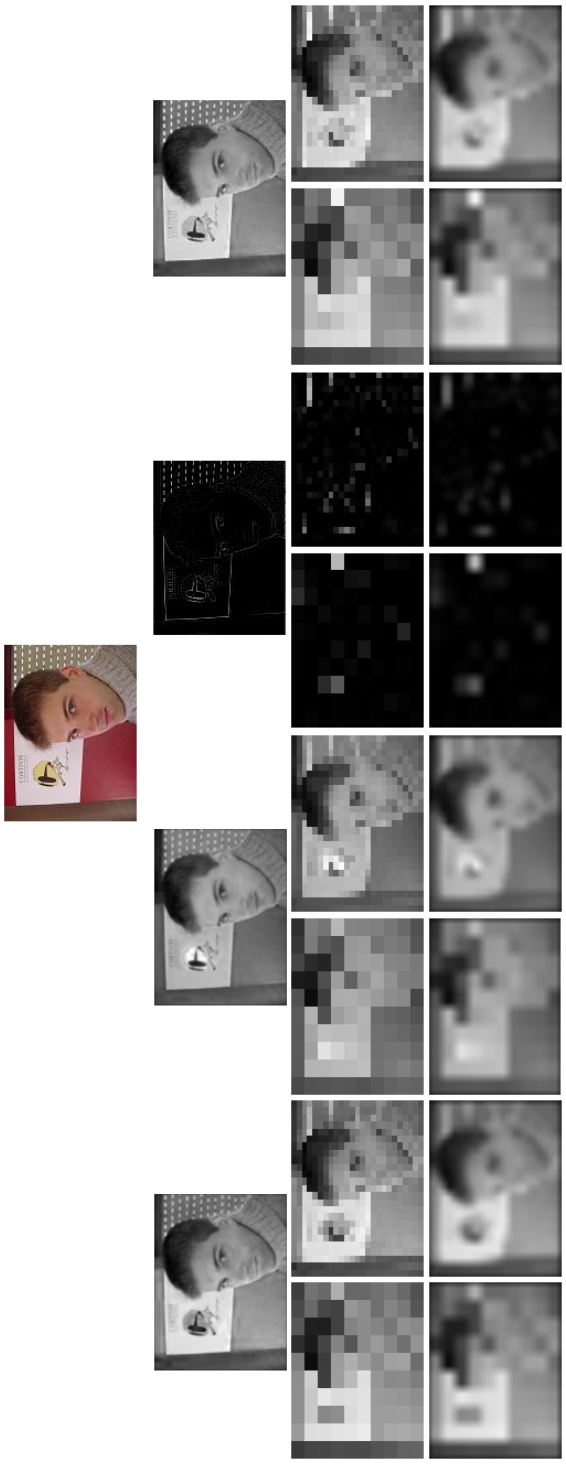

Figura 8.1: Ejemplo de simulación con *Retiner*. A partir de la imagen superior, se han calculado los 3 filtros *DoG* (a, b y c) y la combinación de la ec. 8.1 (d). Para cada filtro, se ha calculado su *pixelado* y su *reconstrucción gaussiana* para una matriz de microelectrodos de 10×10 y 25×25. Máscara de convolución de dimensión 3×3.

Figura 8.2: Comparativa del resultado de aplicar una máscara de convolución de 3×3 y otra de 7×7 sobre el modelo de referencia definido por la ecuación 8.1, tomando esta vez $\sigma_1 = 1.58$, $\sigma_2 = 1.59$ y una matriz de microelectrodos de dimensión 25×25.

8.2.2. Cálculo del modelo en bloques hardware.

Una vez que hemos *sintonizado* todos los parámetros del modelo funcional (que en nuestro caso permanererá inalterado), se utiliza *HSM*, por medio de la ampliación o *plugin HSM–RI* de *Retiner*, para *transportar* la especificación de nuestro sistema (definida anteriormente por el conjunto de filtros de la tabla 8.2), a una especificación *hardware* mediante un diagrama de bloques *CodeSimulink*. De este manera, se pasa del espacio *VHLS* al espacio *HLS* de especificación (Consulte la sección 6.1 y la figura 6.1).

Por tanto, desde *Retiner* y utilizado los parámetros adecuados del *plugin HSM–RI*, se ha calculado automáticamente la arquitectura de bloques hardware de *CodeSimulink* que muestra la figura 8.3.

En ella podemos observar un modelo *CodeSimulink* general para procesar modelos de visión y particularizado para la placa *RC1000 AlphaData*[4]. Dicho modelo general está formado por varios módulos:

- *Entrada de datos*: Una entrada que puede ser tomada de una videocámara (como por ejemplo una *webcam* conectada a un *PC*) o bien[5] generada sintéticamente mediante el bloque *Simulink* conocido por `Uniform Random Number`, que genera en este caso, un flujo de imágenes aleatorias. Ambas opciones generan un flujo contínuo de imágenes *RGB* de 24 bits, con 8 bits por cada canal cromático. Cada *píxel* por tanto viene definido por un entero positivo con un valor en el intervalo $0 \ldots 2^{24} - 1$. Es posible definir otras entradas al modelo de forma manual utilizando otros módulos *Simulink* tales como `From Workspace`, con el que podríamos proporcionar a nuestro sistema una imagen estática.

- *RAM de lectura*: La entrada alimenta un bloque de interfaz[6] con una de las memorias externas de la placa *RC1000* (en este caso el segundo bloque de memoria externa `sim_extRAMread_RC1000_1`). Junto con el bloque anterior de entrada de vídeo, este bloque constituye un *framebuffer* de vídeo digital.

- *Bloques tipo* split *y* unsplit: Posteriormente, la salida de nuestro *framebuffer* alimenta a un bloque tipo *splitter* que divide la señal en sus componentes *R*, *G* y *B*, definiendo para todos los canales el tipo de datos siguiente:

  - ⇝ Datos en punto fijo sin signo.
  - ⇝ Longitud de datos `DW` igual a 8.
  - ⇝ Posición del punto fijo en 0.
  - ⇝ Computación de señales no representables tipo *Wraparound* (que infiere menos hardware).

[4] Placa que, como se comentó en capítulos anteriores, ha sido empleada en las pruebas y los prototipos del trabajo doctoral

[5] Mediante un selector de entradas o `Manual Switch`

[6] De ahí su color anaranjado.

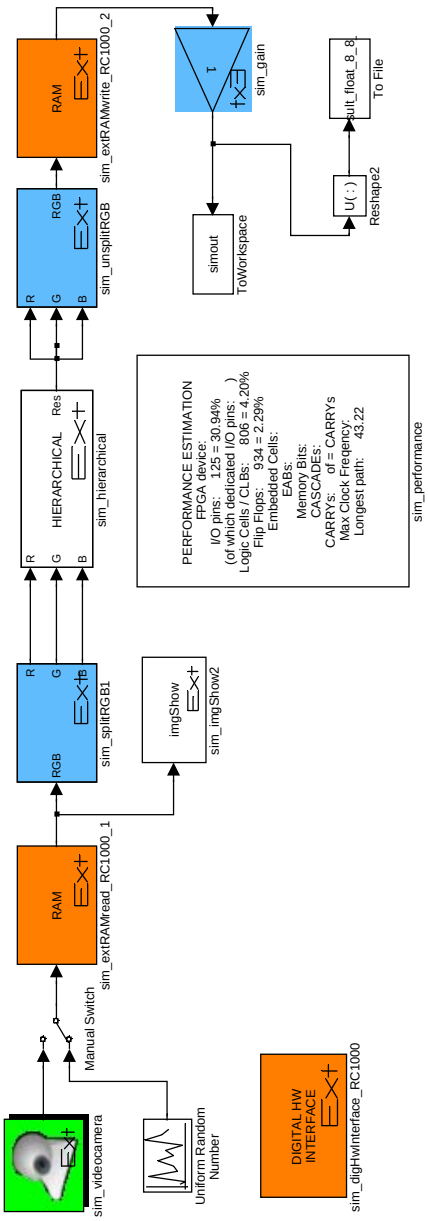

Figura 8.3: Arquitectura en bloques Hw para el procesamiento de sistemas de visión que produce *HSM-RI* para la prueba de prototipos sobre la placa *RC1000*. La entrada puede ser una secuencia de vídeo proporcionada mediante una videocámara o bien valores aleatorios. El procesamiento de la visión propiamente dicho se lleva a cabo en el bloque jerárquico `sim_hierarchical`.

$\rightsquigarrow$ Método de redonde tipo *round*.

Con esto definimos cada canal cromático como enteros positivos de 8 bits (valores entre $0 \ldots 255$). El bloque tipo *unsplitter* actúa de forma inversa al *splitter*, definiendo a la salida un sólo canal de 24 bits de longitud de datos y el resto de propiedades idénticas al enterior.

- *Bloque jerárquico*: Es bloque principal del modelo, y el que define el procesamiento concreto de visión que implementa el modelo. Al contrario que el resto de bloques, el contenido se calcula de forma automática mediante *HSM–RI*.

- *Salida de datos*: De forma similar a la entrada de datos, la salida del sistema alimenta a otra memoria RAM externa de la placa *RC1000*. En este caso el tercer módulo RAM o `sim_extRAMwrite_RC1000_2`.

- *Ganancia adicional*: El sistema generado viene dotado de una ganancia en la etapa final (cuyo valor por defecto es 1)

- *Bloque de interfac digital*: Este bloque es obligatorio en todos los diagramas *CodeSimulink* y guarda toda la información necesaria (*FPGA* y en su caso placa que se va a emplear, asignación patillas, velocidad deseada del sistema, herramienta de síntesis escogida, etc) para el correcto funcionamiento el entorno de simulación y síntesis. En nuestro caso, está particularizada para la tarjeta *AlphaData RC1000* y una *VirtexE 2000*.

- *Módulo de resultados de síntesis*: Este módulo, de nombre `sim_performance`, extrae los datos más significativos de la síntesis lógica del modelo *CodeSimulink* (número de celdas utilizadas, máxima velocidad de procesamiento, etc.) con independencia de la herramienta de síntesis lógica de que se trate (en nuestro caso *Leonardo Spectrum 2004a*).

- *Bloques de visualización de imágenes*: Estos bloques proporcionan el equivalente del comando `imshow` de *Matlab*. Su cometido es representar mediante una imagen sus datos de entrada. Se puede utilizar tanto en el sistema general, como en cualquiera de las conexiones del sistema inferido que veremos a continuación.

La figura 8.4 muestra el modelo de visión que se infiere de forma automática y que se *esconde* tras el bloque `sim_hierarchical` de la figura 8.3[7]. La representación está un poco desordenada en el sentido de que las líneas que modelan las conexiones entre unos bloques y otros, pueden estar superpuestas. Para mejorar la legibilidad del sistema, se ha procedido a reordenar manualmente el sistema tal y como muestra la nueva figura 8.5. Como puede observar, hay un bloque *Simulink* del tipo `ToWorkspace` conectado a cada bloque *CodeSimulink* inferido. Estos bloques se utilizan para grabar la salida de cada bloque en simulación funcional y calcular así el rango dinámico de cada bloque. Consúlte la sección 8.2.3 para más información.

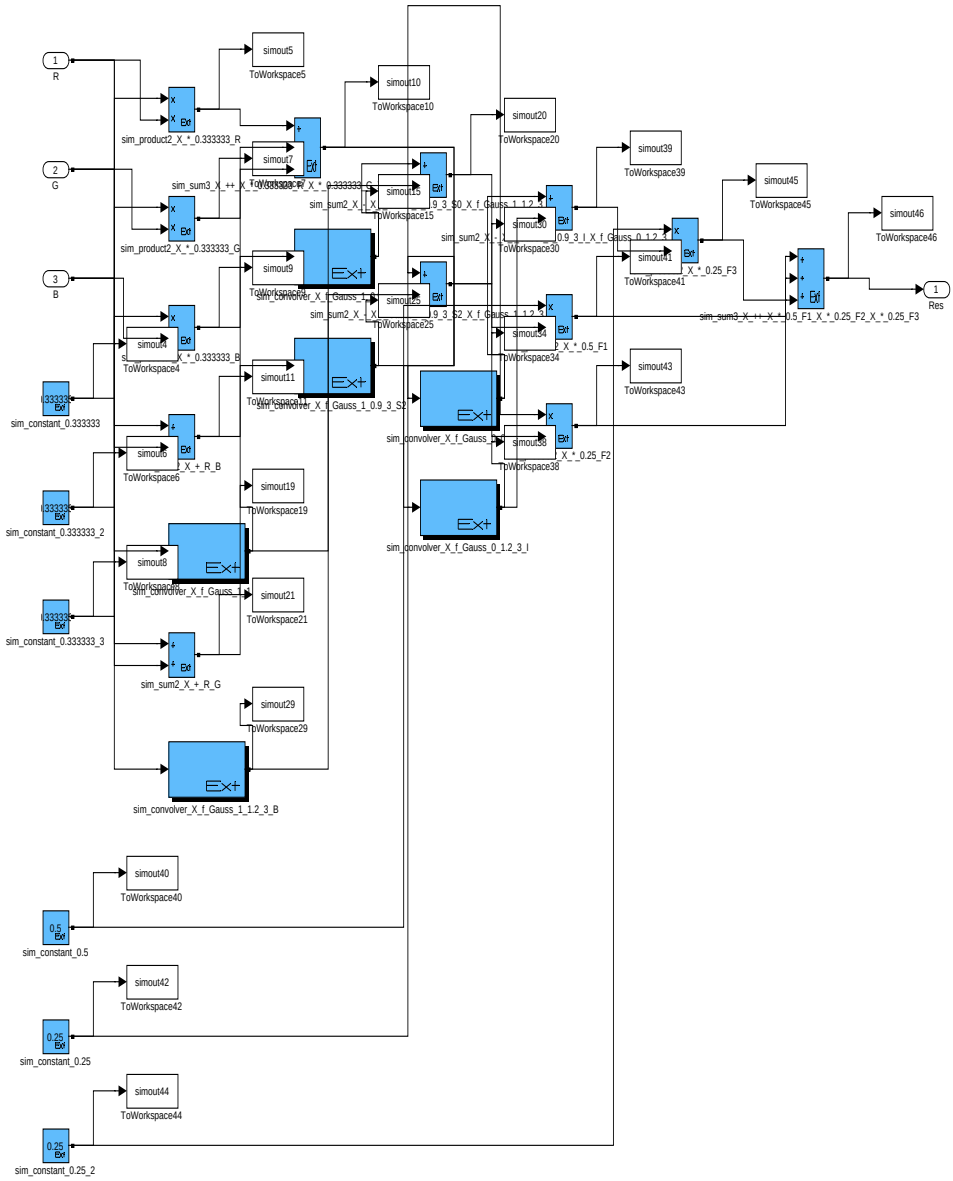

Figura 8.4: Ejemplo del modelo de bloques *CodeSimulink* que infiere la herramienta *HSM–RI* al procesar la ecuación 8.1 que modela el comportamiento retiniano. Este modelo está contenido por el bloque `sim_hierarchical` de la figura 8.3 y como puede observarse define sus tres entradas (`R`, `G` y `B`) y su salida (`Res`).

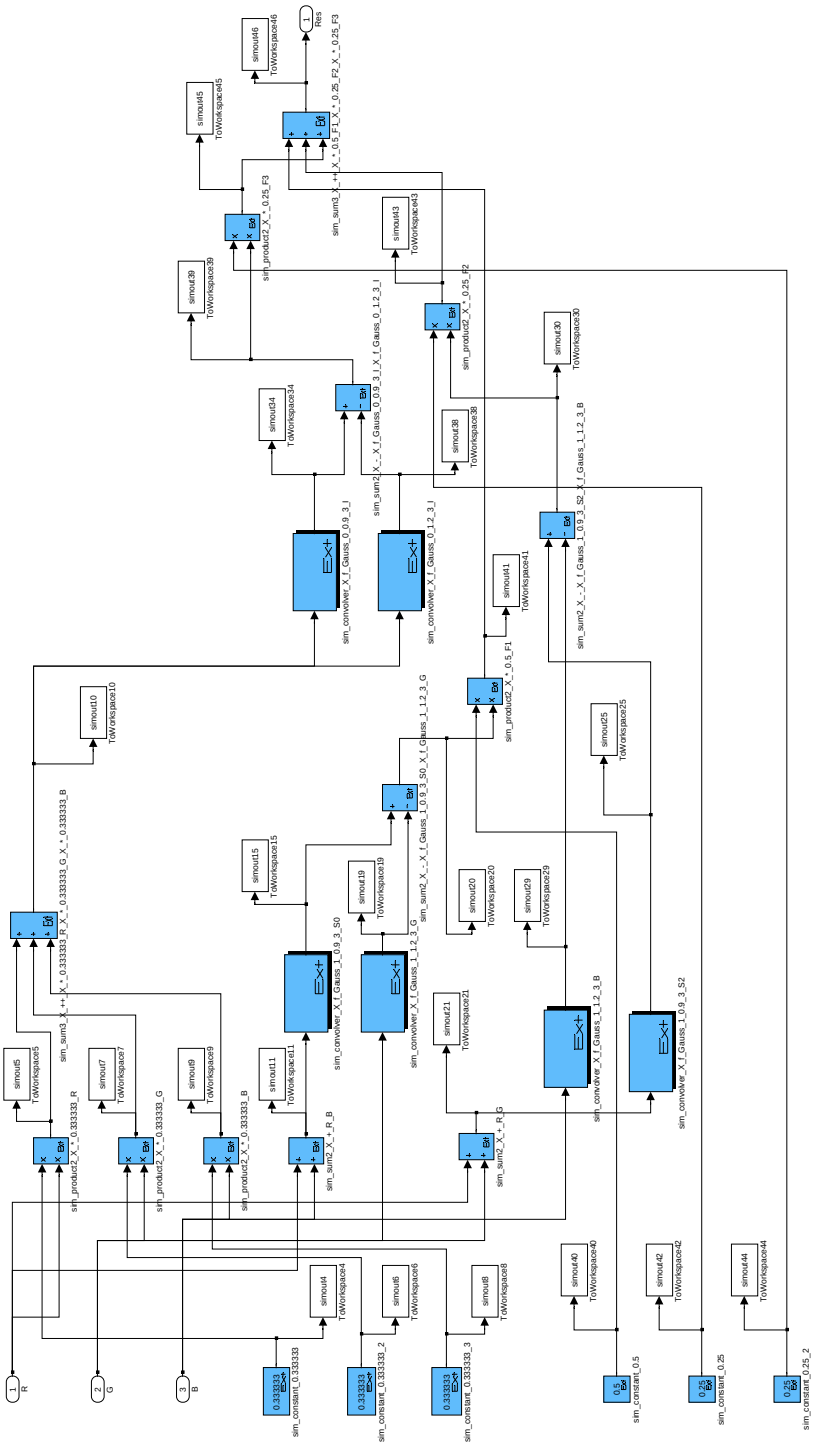

Figura 8.5: Mismo modelo de la figura 8.4 ordenado manualmente para mejorar su comprensión.

El sistema inferido es el mismo modelo que nos ocupa en esta sección. La inferencia del sistema se ha llevado a cabo desactivando la optimización en el espacio de operadores, de ahí que puedan distinguirse seis bloques de convolución y no tan sólo tres, como se obtendrá más adelante activando esta minimización.

Como se detalló en el capítulo 7, el modelo *CodeSimulink* que se infiere puede estar particularizado para realizar simulaciones en *doble precisión* o bien simulaciones compatibles con las restricciones hardware del modelo. De este modo, la simulación en *doble precisión* es similar a la que se obtiene con *Retiner*. Para ilustrar este hecho, la figura 8.6 muestra la captura de pantalla de una sesión de simulación funcional (de alto nivel) utilizando *CodeSimulink* y el sistema total inferido por *HSM–RI*. Pueden distinguirse tres ventanas de visualización de la salida, la primera corresponde a la entrada del sistema, una sencilla *webcam*. La segunda muestra la actuación del filtro `F3` definido prefiamente y que corresponde a una *DoG* del canal de intensidad, que se comporta de form parecida a un detector de bordes (tal y como se observa en la figura). La última ventana de visualización muestra la salida total de nuestro sistema. Como puede comprobrarse, es posible visualizar la salida de cualquier bloque tanto del modelo padre, como del modelo hijo (o bloque jerárquico).

8.2.3. Optimización multi–objetivo

Otra optimización que puede realizar *HSM* es calcular el frente de onda de *Pareto* para los objetivos *área* y *error*, o *velocidad* y *error*. Continuando con el mismo modelo de referencia, se ha activado esta optimización para las dos parejas de posibles objetivos.

HSM selecciona los siguientes parámetros *CodeSimulink* del modelo de visión, que hace variar de forma automática para definir así cada uno de los puntos del diagrama de *Pareto*:

- `SRC`: Que puede ser `unsigned` (o tipo punto fijo sin signo) o `float` (o tipo punto flotante).
- `DW`: Longitud de datos principal del sistema que se ha inferido. Por ejemplo, si se infiere un multiplicador, tendrá este valor en su parámetro `DW`.
- `DWC`: Longitud de datos de los coeficientes de las máscara de convolución.
- `BPC`: Posición del punto decimal de los coeficientes de la máscara. De este modo, los coeficientes de la máscara de convolución se moverán en el intervalo $(2^{DWC} - 1) \cdot 2^{-BPC}$.

Así, para el sistema del ejemplo, *HSM* hace variar estos parámetros conforme a la figura 8.7, de forma que calcula por defecto 24 puntos.

El calculo de la longitud de datos y la posición del punto decimal para cada uno de los bloques que se infieren, se hace por medio de una **adaptación del**

[7] Visible tras hacer *doble–click* sobre el bloque `sim_hierarchical`

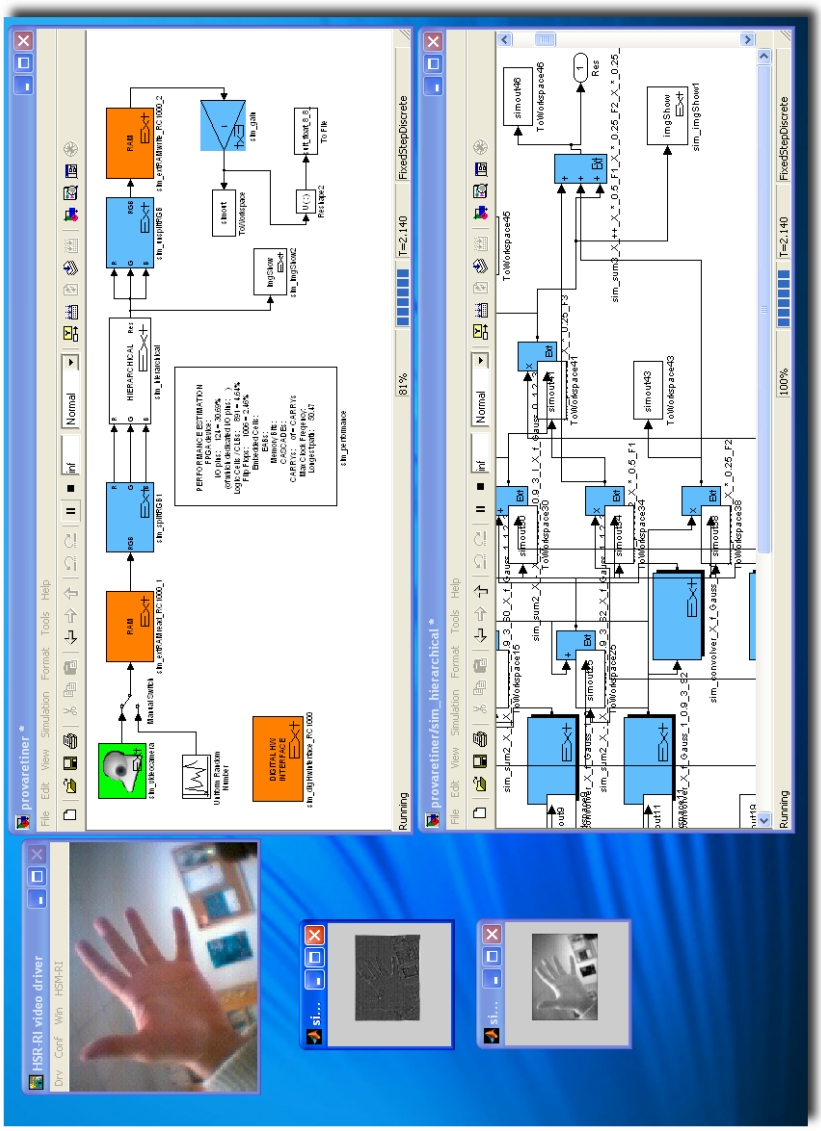

Figura 8.6: Captura de pantalla que muestra una sesión de simulación funcional del modelo de referencia utilizando *CodeSimulink*.

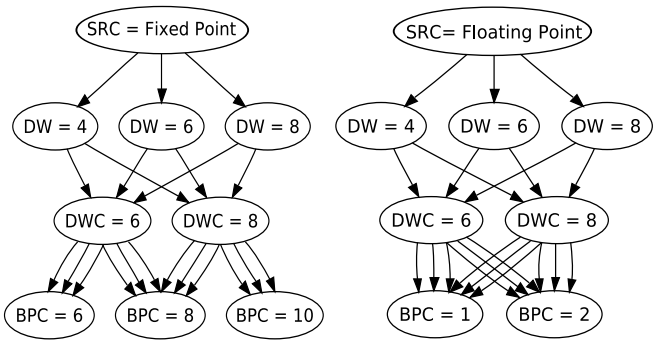

Figura 8.7: Conjunto de valores que toman los parámetros de la simulación.

rángo dinámico individual para cada bloque. Así, el valor `BPC` los datos de la salida de cada bloque, dependerá del parámetro `BPC` que varía *HSM*, y este a su vez del Log_2 (logaritmo en base 2) del rango dinámico del bloque.

Una vez lanzada esta optimización, *HSM* calcula automáticamente los frentes de *Pareto* de las figuras 8.8 (para los objetivos área y error) y 8.9 (para los objetivos velocidad y error).

En el eje de ordenadas se expresa el error *RMS* o error cuadrático medio que se obtiene al comparar la simulación funcional y simulación basada en hardware (definida por parámetros del punto del diagrama que se trate).

Como puede comprobarse, el punto de mejor relación área-error es el punto 17, que tiene las siguientes características:

RMS(Error)	1.0208	`SRC`	coma flotante
`DW`	6	`CLBs`	934 (4.2 %)
`DWC`	6	`FF`	934 (2.29 %)
`BPC`	1	Período	$43.22ns$

Tabla 8.3: Datos de la síntesis del sistema calculada mediante *Leonardo Spectrum 2004* equivalentes al mejor punto de síntesis (número 17) en términos del compromiso entre área ocupada por el circuito (*FPGA*) y el error cuadrático medio que se produce se obtiene. `FF` = número de flip flops.

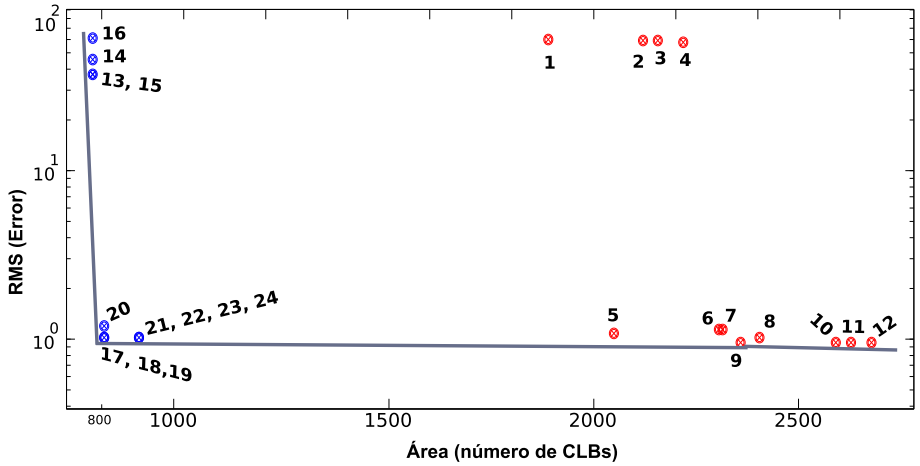

Figura 8.8: Frente de *Pareto* para los objetivos error y área, resultado de sintetizar el ejemplo que nos ocupa.

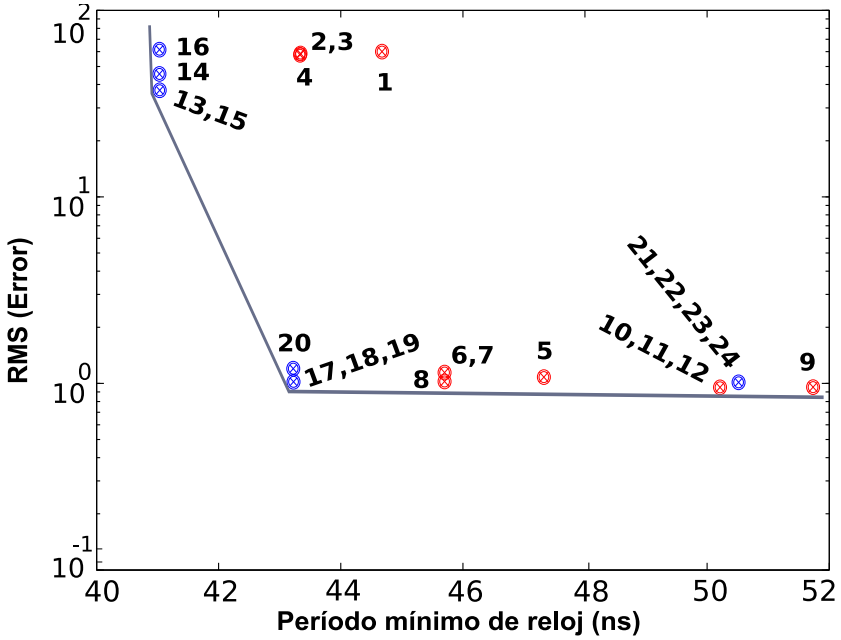

Figura 8.9: Frente de *Pareto* para los objetivos error y mínimo período de reloj (máxima velocidad de proceso), resultado de sintetizar el ejemplo que nos ocupa.

Síntesis	SRC	DW	DWC	BPC	**Síntesis**	SRC	DW	DWC	BPC
1.	1	4	6	6	13.	2	4	6	1
2.	1	4	6	8	14.	2	4	6	2
3.	1	4	8	8	15.	2	4	8	1
4.	1	4	8	10	16.	2	4	8	2
5.	1	6	6	6	17.	2	6	6	1
6.	1	6	6	8	18.	2	6	6	2
7.	1	6	8	8	19.	2	6	8	1
8.	1	6	8	10	20.	2	6	8	2
9.	1	8	6	6	21.	2	8	6	1
10.	1	8	6	8	22.	2	8	6	2
11.	1	8	8	8	23.	2	8	8	1
12.	1	8	8	10	24.	2	8	8	2

En este caso, el punto de mejor relacción velocidad–error es también el número 17.

El cálculo del frente de *Pareto* y del punto con el mejor compromiso entre dos objetivos dados, se calcula también de forma automática. Las gráficas (en su origen gráficas de *Matlab*) se han retocado con objeto de mejorar la legibilidad, ya que en los originales (calculados automáticamente) se solapan las etiquetas de los puntos cuyas coordenadas coinciden, haciendo muy difícil distinguir unos puntos de otros.

8.2.4. Optimización en el espacio de operadores.

Tal y como se vio en el capítulo 7, toda combinación lineal de una serie de filtros convolutivos, se puede expresar como una combinación lineal de tantos filtros convolutivos como variables independientes se utilicen en los mismos. En este caso, tenemos tres variables independientes (R,G y B) y se ha reducido el número total de filtros de convolución necesarios seis a tres. Nótese que en el caso del modelo que nos ocupa, el filtro convolutivo que representa cada gaussiana es el componente del sistema que más área ocupa en el sistema hardware final.

Activando la optimización del modelo en el espacio de operadores, *HSM* reescribe internamente la ecuación 8.1 según la nueva expresión 8.5:

$$\begin{aligned} Modelo \quad = \quad & \frac{1}{2}(\Gamma_1 * R + \Gamma_1 * B - \Gamma_2 * G) + \\ & \frac{1}{4}(\Gamma_1 * R + \Gamma_1 * G - \Gamma_2 * B) + \\ & \frac{1}{4}(\Gamma_1 * \frac{R}{3} + \Gamma_1 * \frac{G}{3} + \Gamma_1 * \frac{B}{3} - \\ & \Gamma_2 * \frac{R}{3} - \Gamma_2 * \frac{G}{3} - \Gamma_2 * \frac{B}{3}) \end{aligned} \tag{8.5}$$

siendo $\Gamma_1 = Gauss(0.9, 3, \cdot)$ la máscara de convolución 3×3 asociada a la gaussiana con $\sigma = 0.9$ y Γ_2 ídem para la gaussiana con parámetro $\sigma = 1.2$.

Finalmente, minimizando la ecuación 8.5, *HSM* da como resultado una minimización del sistema completo dada por la ecuación 8.6:

$$Modelo = \kappa_R \star R + \kappa_G \star G + \kappa_B \star B \tag{8.6}$$

con

$$\kappa_R = \frac{2}{3} \cdot \Gamma_1 - \frac{1}{12} \cdot \Gamma_2 \tag{8.7}$$

$$\kappa_G = \frac{1}{3} \cdot \Gamma_1 - \frac{7}{12} \cdot \Gamma_2 \tag{8.8}$$

$$\kappa_B = \frac{7}{12} \cdot \Gamma_1 - \frac{1}{3} \cdot \Gamma_2 \tag{8.9}$$

y donde $\star$ representa el operador convolución.

Se disminuye por tanto el número total de bloques gaussiana, y en general de filtros de convolución por un factor 2 en este caso.

Las figuras 8.10 y 8.11 muestran la nueva configuración del bloque jerárquico del ejemplo, al activar esta optimización. La primera de ellas es una captura original, que está ordenada manualmente en la segunda. Haciendo una rápida comparación visual, es evidente el ahorro en área que se produce y la enorme simplificación que se lleva a cabo.

8.2.5. Comparativa entre modelos realizados con *Handel-C* y con HSM.

En la cita bibliográfica [45] puede consultarse una realización del mismo modelo de retina definido por la ecuación 8.1, empleando máscaras de convolución de dimensión 7×7. Dicha realización se llevó a cabo mediante una descripción del sistema en *Handel-C* y una posterior síntesis con la herramienta *DK* de *Celoxica*. El modelo de retina que se sintetiza viene descrito por la ecuación 8.10.

$$\begin{aligned} Modelo = \frac{1}{4} \cdot [2\, DoG(0.9, 1.2, 3, R+B, G) + DoG(0.9, 1.2, 3, R+G, B) + \\ DoG(0.9, 1.2, 3, I, I)] \end{aligned} \tag{8.10}$$

Teniendo en cuenta de que los parámetros y las entradas del prototipo son idénticas a las que se han visto en esta sección (es decir imagen de entrada de dimensión 120×160, placa *RC1000* de *Celoxica* con la *FPGA VirtexE 2000*, etc) y habida cuenta de la diferencia entre las dos herramientas en cuanto a la entrada de descripción del sistema y el distinto proceso de optimización, se procede a comparar ambos modelos mediante la tabla 8.4.

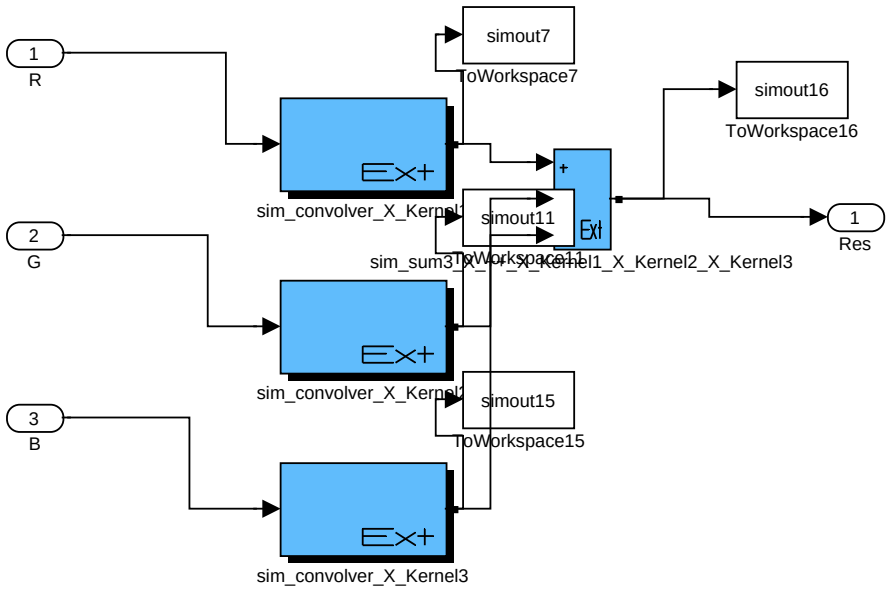

Figura 8.10: Ejemplo del modelo de bloques *CodeSimulink* que infiere la herramienta *HSM–RI* al aplicar todas las optimizaciones posibles.

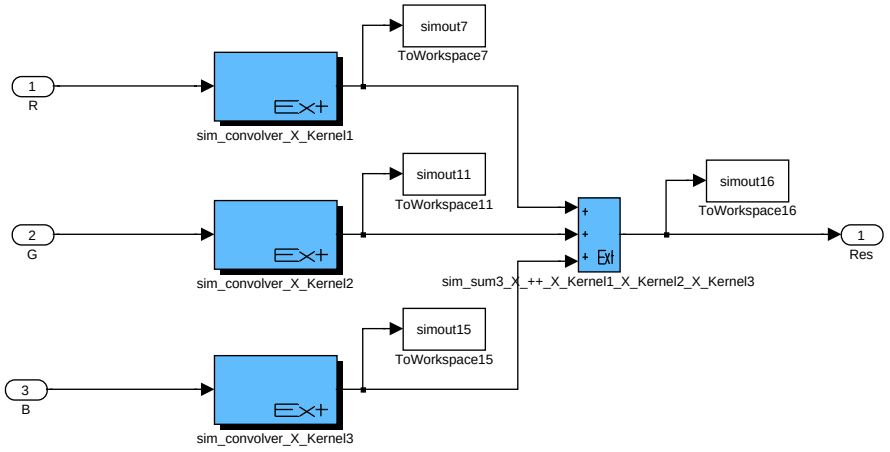

Figura 8.11: Mismo modelo de la figura 8.10 ordenado manualmente para mejorar su comprensión.

	Handel-C & DK	*HSM*
Área (*CLBs*)	9181	993
Período de reloj	$15.281ns$	46.61
Tipo de datos	Enteros	Punto flotante
Coeficientes de la máscara	Potencias de 2	reales

Tabla 8.4: Tabla comparativa blablabla

8.3. Contraste de resultados biológicos y sintéticos.

A modo de ejemplo, la figura 8.12 muestra una comparación entre las respuestas que, ante un mismo estímulo visual, ofrecen una retina biológica y otra sintetizada siguiendo la combinación de filtros de la ecuación 8.1.

El gráfico muestra, en la parte superior, los instantes en que la retina es expuesta a un estímulo visual definido como *flash* global de luz blanca, a intervalos periódicos (iluminación blanca sobre toda la retina durante un cierto período de tiempo). En el caso del tejido biológico, la respuesta se recoge en la gráfica de la parte central. Para construir esta representación se han registrado, mediante una matriz de microelectrodos de *Utah*, las señales provenientes de un área concreta de la retina expuesta (el área donde se sitúa la matriz). Estas señales se someten a un proceso de acondicionamiento, para eliminar ruido, y clasificar el origen de cada respuesta recogida en un determinado electrodo. Tras el acondicionamiento de señal y la clasificación de unidades de cada electrodo[8], se ha realizado una selección para mostrar en la gráfica aquellas unidades que exhiben un comportamiento del tipo triada fotorreceptores–bipolares–ganglionares tipo *ON*. Como se puede observar, los canales de la matriz registran una serie de *disparos* que ocurren breves instantes después de que se comience a aplicar el *flash*, y que se suceden mientras el estímulo dura, e incluso continúan breves instantes después de que se extinga.

Full-field flashing stimulation of a retina: (top) stimulus occurrences, (middle) in vivo recording from a rabbit retina, and (bottom) output of Retiner simulation for this stimulus.

Por otra parte, el modelo de retina definido mendiante *Retiner*, según la citada ecuación, produce, ante el mismo estímulo, la respuesta recogida en la parte inferior de la gráfica. Para este modelo se ha seleccionado con *Retiner*, el cálculo de la respuesta análoga *ON*. Como se puede observar, dicha respuesta es bastante similar a la que ofrece la retina biológica, con la salvedad de la regularidad propia de un sistema artificial.

Teniendo en cuenta que el tejido biológico de la retina presenta una actividad de disparos esporádicos en reposo, es obvio que la respuesta no podrá ser idéntica a la del sistema artificial. Sin embargo, el parecido en la respuesta es indicativo de que el modelo presenta un comportamiento similar al de una retina natural ante el mismo estímulo luminoso.

[8]Mediante una separación de fuentes, se llega a distinguir la contribución de las neuronas que excitan cada electrodo. Cada tren de *spikes* asociado a un electrodo dado, lleva el nombre de *unidad*

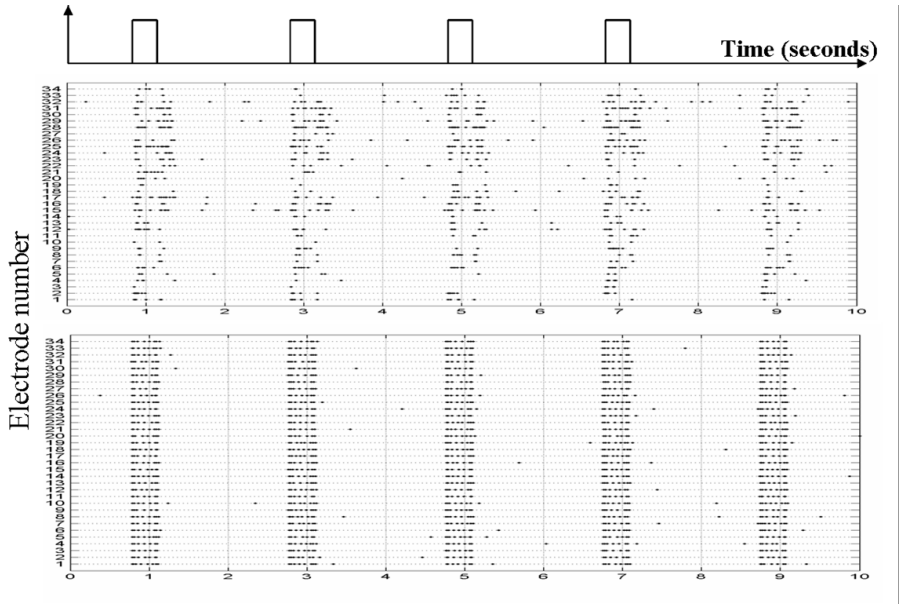

Figura 8.12: Estimulación mediante destellos de luz blanca, de una retina: La gráfica de arriba muestra la secuencia de estimulación, la del medio el resultado biológico real obtenido con una retina de conejo. Abajo se añade la salida que produce *Retiner* frente al mismo estímulo.

Todo el proceso de definición del modelo, simulación, selección de las componentes biológicas más representativas y la posterior visualización, ha sido realizada con *Retiner*. Por último, la *sintonización* o búsqueda de parámetros del modelo en alto nivel que vayan aproximando sucesivamente ambos resultados, configura una realimentación del modelo de retina que cierra su bucle de diseño.

Capítulo 9

Conclusiones y principales aportaciones

En el contexto de los sistemas artificiales para procesamiento de información visual, especialmente cuando es necesario un funcionamiento en tiempo real mientras se interactúa visualmente con el entorno, y en equipos portátiles, es esencial disponer de soluciones integradas en circuitos dedicados, que procesen el flujo de imágenes con la celeridad requerida. Si añadimos como requisito adicional la posibilidad de reconfiguración, las *FPGA* se convierten en los candidatos ideales para implementar dichos circuitos. Ofrecen el grado de paralelismo fino necesario, a la vez que unas características contenidas de coste y consumo de potencia.

La implementación eficiente de un sistema de visión en hardware reconfigurable (en chips *FPGA*) es sin embargo una tarea ardua y susceptible de errores; siendo de gran ayuda disponer de herramientas de simulación y de síntesis automática que partan de un nivel de abstracción lo más alto posible.

El campo de aplicación que motivó este trabajo de tesis fue el desarrollo de sistemas de procesamiento para neuroprótesis visuales a nivel cortical, en el contexto del proyecto europeo *CORTIVIS*. No obstante, los modelos y herramientas obtenidas tienen un campo de aplicación más amplio, que abarca el desarrollo de aproximaciones neuromórficas a sistemas artificiales de visión, el diseño de modelos bioinspirados de visión por computador, el modelado funcional de retinas biológicas y la caracterización de patologías o deficiencias visuales.

En este contexto de aplicaciones, este trabajo de tesis doctoral presenta las siguientes aportaciones:

1. Se ha definido una plataforma de diseño hardware/software para el desarrollo completo de sistemas bioinspirados de visión. El hardware consiste en un computador *PC* con una tarjeta aceleradora basada en *FPGA* de altas prestaciones. El software combina módulos definidos sobre *MATLAB/Simulink* y herramientas de síntesis de bajo nivel. En su conjunto, la plataforma permite la especificación, simulación funcional, síntesis de circuito y simulación con las restricciones del hardware, así como la

verificación experimental del sistema con secuencias de imágenes reales.

2. Se ha definido un modelo con una arquitectura de referencia que abstrae las características funcionales esenciales de las retinas biológicas, tales como el realce espacial y temporal locales, el contraste de color, y las diferencias de comportamiento centro-periferia de la retina. La arquitectura básica de referencia consiste en una combinación de filtros de imagen, fácilmente ampliable, cuya salida es codificada en impulsos mediante módulos integrador-disparador.

3. En un trabajo conjunto con otros miembros del equipo investigador de *CORTIVIS*, se ha implementado el entorno software *RETINER*, una aplicación definida sobre *MATLAB*, que permite la especificación ágil y simulación de modelos de retinas, así como la comparación con registros nerviosos multielectrodo obtenidos directamente de retinas biológicas.

4. Se ha desarrollado un conjunto de módulos hardware paramétricos descritos en el lenguaje *VHDL* estándar del *IEEE*, que cumplen con las especificaciones definidas para los bloques *CodeSimulink*, y que amplían las posibilidades de esta herramienta de co–diseño desarrollada en el Politécnico de Turín, al campo de los sistemas hardware de visión.

5. Se ha ideado e implementado un sintetizador de alto nivel, que hemos denominado *HSM*, capaz de transformar y optimizar una descripción funcional procedente de *RETINER* en una descripción estructural de flujo de datos basada en los bloques *CodeSimulink*.

6. Se ha comprobado que los resultados de síntesis en hardware obtenidos son válidos para distintas herramientas de síntesis de más bajo nivel (sintetizadores lógicos *Leonardo Spectrum*, *Quartus*, e *ISE XST*) y para plataformas *FPGA* de distintos fabricantes (*Xilinx* y *Altera*). Igualmente, dadas las características del código que se genera, las descripciones de circuito son también válidas para su compilación en bibliotecas de celdas para circuitos integrados *VLSI* específicos.

7. Las características de área y velocidad del circuito resultante de la síntesis dependen de la precisión de cálculo exigida. Un análisis de estas dependencias con ejemplos concretos de síntesis nos ha permitido fijar las bases para un procedimiento automático de exploración del espacio de diseño, basado en la selección de un conjunto óptimo de parámetros de los bloques hardware, ligados a los tamaños de datos y al tipo de cálculo aritmético empleado.

A lo largo de la realización de la presente tesis doctoral, y en relación con los resultados y aportaciones de la misma, se han generado (de forma total o parcial) las siguientes publicaciones:

Relacionadas con los capítulos 4, 6 y 7.

- Antonio Martínez, Leonardo M. Reyneri, Francisco J. Pelayo, Samuel F. Romero, Christian A. Morillas, Begoña Pino, *Automatic Generation of Bio-inspired retina-like Processing Hardware*, *Lecture Notes in Computer Science*, Ed. Springer, número 3512, págs. 527 – 533, año 2005, ISSN: 0302–9743, ISBN: 3-540-26208-3.

- Antonio Martínez Álvarez, Leonardo Maria Reyneri, Francisco J. Pelayo Valle, *Automatic synthesis of data-flow systems using a high level codesign tool. Application to vision processors.*. Aceptado para *The 13th IEEE Mediterranean Electrotechnical Conference* (*MELECON*'2006).

- Antonio Martínez, Leonardo M. Reyneri, Francisco J. Pelayo, Samuel F. Romero, Christian A. Morillas, Begoña Pino, *Automatic Generation of Bio-inspired retina-like Processing Hardware*, *International Work-Conference on Artificial Neural Networks (IWANN'2005)*, Vilanova i la Geltrú, (Barcelona), 8 – 10 Junio de 2005, ISSN: 0302-9743, ISBN: 3-540-26208-3.

- A. Martínez, F.J. Pelayo, C. Morillas, S. Romero, R. Carrillo, B. Pino, *Generador automático de sistemas bioinspirados de visión en hardware reconfigurable*, IV Jornadas de Computación Reconfigurable y Aplicaciones (*JCRA*'04), Barcelona, 13 – 15 de Septiembre 2004, ISSN: 84-688-7667-4.

- Antonio Martínez, Francisco J. Pelayo, Christian A. Morillas, Leonardo M. Reyneri, Samuel Romero, *Automatic synthesis of vision processors on reconfigurable hardware*, *V Jornadas de Computación Reconfigurable y Aplicaciones* (*JCRA*'05), dentro del *I Congreso Español De Informática* (*CEDI*'2005), 13 – 16 Septiembre 2005, Granada. páginas 179 – 184, ISBN: 84-9732-439-0.

Relacionada con el capítulo 5.

- L. Sousa, P. Tomás, F. Pelayo, A. Martínez, C. A. Morillas, S. Romero, *Bioinspired Stimulus Encoder for Cortical Visual Neuroprostheses*. Capítulo del libro *New Algorithms, Architectures, and Applications for Reconfigurable Computing*, págs. 279 – 290, Ed. *Springer*, Editores P. Lysaght y W. Rosenstiel, año 2005, ISBN: 1-4020-3127-0.

- Samuel Romero, Francisco J. Pelayo, Christian A. Morillas, Antonio Martínez, Eduardo Fernández, *Reconfigurable Retina-like Preprocessing Platform for Cortical Visual Neuroprostheses*. Capítulo del libro: *Neural Engineering: Neuro-Nanotechnology - Biorobotics, Artificial Implants and Neural Prosthesis*, Vol 3, Ed. *Metin Akay*, *IEEE Press Series on Biomedical Engineering*, (en imprenta), ISBN: 0-471-68023-0.

- F.J. Pelayo, S. Romero, C. Morillas, A. Martínez, E. Ros, E. Fernández, *Translating image sequences into spikes patterns for cortical neuro-stimulation*,

revista *Neurocomputing*, (*Computational Neuroscience*), número 58-60, págs. $885 - 892$, año 2004, Editorial *Elsevier*, ISSN: 0925-2312.

- Christian A. Morillas, Samuel F. Romero, Antonio Martínez, Francisco J. Pelayo, Eduardo Ros, Eduardo Fernández, *A Design Framework to Model Retinas*, revista *Biosystems, Computational Neuroscience*, Ed. *Elsevier*, año 2004, ISSN: 0303-2647.

- L. Sousa, P. Tomás, F. J. Pelayo, A. Martínez, C. Morillas, S. Romero, *A FPL bioinspired visual encoding system to stimulate cortical neurons in real time*, revista *Lecture Notes in Computer Science*, ed. *Springer*, número 2778, págs $691-700$, año 2004, ISSN: 0302-9743, ISBN: 3-540-40822-3.

- Christian A. Morillas, Samuel F. Romero, Antonio Martínez, Francisco J. Pelayo, Eduardo Fernández, *A Computational Tool to Test Neuromorphic Encoding Schemes for Visual Neuroprostheses*, revista *Lecture Notes in Computer Science*, ed. *Springer*, número 3512, págs $510 - 517$, año 2005, ISSN: 0302-9743, ISBN: 3-540-26208-3.

- F. J. Pelayo, A. Martínez, S. Romero, C. Morillas, E. Ros, E. Fernández, *Cortical Visual Neuroprosthesis for the Blind: Retina-like Software/Hardware Preprocessor*, congreso *1st International IEEE EMBS Conference on Neural Engineering*, Isla de Capri (Italia), $20 - 22$ Marzo 2003, ISBN: 0-7803--7579-3.

- F. J. Pelayo, A. Martínez, C. Morillas, S. Romero, L. Sousa, P. Tomás, *Retina-like Processing and Coding Platform for Cortical Neuro-stimulation*, *25th Annual International Conference of the IEEE EMBS*, Cancún (México), $17 -$ 21 Septiembre, 2003, ISBN: 0-7803-7790-7.

- Romero, C. Morillas, A. Martínez, F. J. Pelayo, E. Fernández, *Models and tools for testing visual neuroprosthesis*, *4th Forum of European Neuroscience (FENS)*, Lisboa (Portugal), 2004.

- E. Fernández, C. Morillas, S. Romero, A. Martínez, F. Pelayo, *Neuroengineering Tools for the Design and Test of Visual Neuroprostheses*, congreso *Annual Meeting of The Association for Research in Vision and Ophtalmology*, Fort Lauderdale (Florida), $1 - 5$ Mayo 2005, ISSN: 1552-5783, E-Abstract 1483 (publicado en un número electrónico de *Investigative Ophthalmology & Visual Science*, Volumen 46, Mayo 2005, http://abstracts.iovs.org/cgi/content/abstract/46/5/1483).

- S. Romero, A. Martínez, Ch. A. Morillas, F. J. Pelayo, E. Fernández, *Visual Information Pre-Processing System for Visual Neuroprostheses*, *XI Congreso Nacional de la Sociedad Española de Neurociencia*. Torremolinos (Málaga). Abstract publicado en el suplemento 2 del volumen 41 (2005) número 6 de la *Revista de Neurología*. ISSN: 0210-0010.

- S. Romero, C. Morillas, A. Martínez, F. Pelayo, E. Fernández, *A Research Platform for Visual Neuroprostheses*, *Simposio de Inteligencia Computacional*, *SICO'*2005 (*IEEE Computational Intelligence Society, SC*), págs. 357 − 362, Granada 13 − 16 Septiembre 2005, ISBN: 84-9732-444-7.
- A. Martínez, S. Romero, E. Ros, A. Prieto, F.J. Pelayo, *Implementación de hardware reconfigurable de un modelo de retina*, congreso *II Jornadas sobre Computación Reconfigurable y Aplicaciones* (*JCRA'*2002), págs. 97−101, ISBN:-84-699-9448-4, Almuñécar (Granada), 18 − 20 Septiembre 2002.

Capítulo 10

Conclusions and main contributions

In the context of artificial systems for visual information processing, especially when real-time performance is required and in portable applications, the availability of custom integrated solutions for processing the stream of images is essential with the required speed. In addition, if reconfigurability is a requirement, *FPGAs* becomes the ideal candidates to implement these circuits. *FPGAs* offer enough degree of fine–grain parallelism and, at the same time, a low cost and a reduced power consumption.

However, the implementation of vision systems by means of reconfigurable hardware (*FPGA* devices) is a hard task and susceptible of multiple errors; being of great help to count with simulation and automatic synthesis tools which allows the description of the system from a high abstraction level. The main motivation for this thesis work was the development visual neuroprothesis at cortical level, and has been developed in the context of the european research project *CORTIVIS*. However, the proposed models and tools have a wide field of applications, covering from developing neuromorphic approaches to artificial vision systems, designing bioinspired models of computer vision, functional modeling of biologic retinas and characterization visual patologies.

In this applications environment, the presented work shows the following contributions:

1. A hardware/software design platform has been defined for developing of bioinspired vision systems. The hardware is composed by a PC workstation with a co-processor board populated with a high performance *FPGA*. The software combines different modules described with Matlab/Simulink and low–level synthesis tools. The platform allows the especification, functional and post-layout simulation, hardware synthesis and experimental verification of the system with real images.

2. A reference architecture has been defined. This model abstracts the essential functional features of biological retinas, like local spacial and temporal enhancement, color contrast, and the different behaviour between

the centre and peripheral retina. The reference architecture is composed by a mixed of image filters, easily extendable, whose output is coded into impulses by means of integrative–trigger modules.

3. Jointly wiht other research members of *CORTIVIS*, has been implemented the software environment *RETINER*. This application, which has been built using *MATLAB* tool, allows a flexible especification and simulation of retina models and in the same way the comparation of biological multielectrode retina records.

4. A library of parametized hardware modules has been developed using *VHDL*. These modules fulfill the specifications of the *CodeSimulink* blocks (a co–design tool developed in the Polithecnic of Turin) and enlarge their possibilities to the specific domain of vision systems.

5. A high level synthesizer has been designed and implemented (called HSM). *HSM* tool transforms and optimizes a *RETINER* functional description into an estructural description of data flow based on CodeSimulink blocks.

6. The results of the high-level synthesis have been validated for several low-level synthetizers (*Leonardo Spectrum*, *Quartus* and *ISE XST*) and *FPGA* devices of different companies (*Xilinx* and *Altera*). In addition, the descriptions obtained are also valid for implementation on custom *VLSI* circuits.

7. The area and speed features of the resulting circuits depend of the accuracy constrains. The analysis of concrete examples of these dependencies has allowed to stablish the basis of an automatic procedure for design space exploration, based on the selection of an optimized set of parameters related to the size of data and the artihmetic operations perform by the hardware blocks.

In the work of the presented dissertation, and related with the results and main contributions, we have generated the next list of publications:

Related with chapters 4, 6 and 7.

- Antonio Martínez, Leonardo M. Reyneri, Francisco J. Pelayo, Samuel F. Romero, Christian A. Morillas, Begoña Pino, *Automatic Generation of Bio-inspired retina-like Processing Hardware*, *Lecture Notes in Computer Science*, Ed. Springer, number 3512, pp. 527 – 533, year 2005, ISSN: 0302-9743, ISBN: 3-540-26208-3.

- Antonio Martínez Álvarez, Leonardo Maria Reyneri, Francisco J. Pelayo Valle, *Automatic synthesis of data-flow systems using a high level codesign tool. Application to vision processors.*. Accepted for *The 13th IEEE Mediterranean Electrotechnical Conference* (*MELECON*'2006).

- Antonio Martínez, Leonardo M. Reyneri, Francisco J. Pelayo, Samuel F. Romero, Christian A. Morillas, Begoña Pino, *Automatic Generation of Bio-inspired retina-like Processing Hardware*, *International Work-Conference on Artificial Neural Networks (IWANN'2005)*, Vilanova i la Geltrú, (Barcelona), June 8 – 10, 2005, ISSN: 0302-9743, ISBN: 3-540-26208-3.

- A. Martínez, F.J. Pelayo, C. Morillas, S. Romero, R. Carrillo, B. Pino, *Generador automático de sistemas bioinspirados de visión en hardware reconfigurable*, IV Jornadas de Computación Reconfigurable y Aplicaciones (*JCRA'*04), Barcelona, 13 – 15 September, 2004, ISSN: 84-688-7667-4.

- Antonio Martínez, Francisco J. Pelayo, Christian A. Morillas, Leonardo M. Reyneri, Samuel Romero, *Automatic synthesis of vision processors on reconfigurable hardware*, *V Jornadas de Computación Reconfigurable y Aplicaciones* (*JCRA'*05), 13 – 16 September 2005, Granada. pp. 179 – 184, ISBN: 84-9732-439-0.

Related with chapter 5.

- L. Sousa, P. Tomás, F. Pelayo, A. Martínez, C. A. Morillas, S. Romero, *Bioinspired Stimulus Encoder for Cortical Visual Neuroprostheses*. Charpter of the book *New Algorithms, Architectures, and Applications for Reconfigurable Computing*, pp. 279–290, Ed. *Springer*, Editores P. Lysaght y W. Rosenstiel, year 2005, ISBN: 1-4020-3127-0.

- Samuel Romero, Francisco J. Pelayo, Christian A. Morillas, Antonio Martínez, Eduardo Fernández, *Reconfigurable Retina-like Preprocessing Platform for Cortical Visual Neuroprostheses*. Chapter of the book: *Neural Engineering: Neuro-Nanotechnology - Biorobotics, Artificial Implants and Neural Prosthesis*, Vol. 3, Ed. *Metin Akay*, *IEEE Press Series on Biomedical Engineering*, (in press), ISBN: 0-471-68023-0.

- F.J. Pelayo, S. Romero, C. Morillas, A. Martínez, E. Ros, E. Fernández, *Translating image sequences into spikes patterns for cortical neuro-stimulation*, revista *Neurocomputing*, (*Computational Neuroscience*), number 58-60, pp. 885 – 892, year 2004, Ed. *Elsevier*, ISSN: 0925-2312.

- Christian A. Morillas, Samuel F. Romero, Antonio Martínez, Francisco J. Pelayo, Eduardo Ros, Eduardo Fernández, *A Design Framework to Model Retinas*, *Biosystems, Computational Neuroscience*, Ed. *Elsevier*, year 2004, ISSN: 0303-2647.

- L. Sousa, P. Tomás, F. J. Pelayo, A. Martínez, C. Morillas, S. Romero, *A FPL bioinspired visual encoding system to stimulate cortical neurons in real time*, *Lecture Notes in Computer Science*, ed. *Springer*, number 2778, pp. 691–700, year 2004, ISSN: 0302-9743, ISBN: 3-540-40822-3.

- Christian A. Morillas, Samuel F. Romero, Antonio Martínez, Francisco J. Pelayo, Eduardo Fernández, *A Computational Tool to Test Neuromorphic*

Encoding Schemes for Visual Neuroprostheses, revista *Lecture Notes in Computer Science*, ed. *Springer*, número 3512, pp. 510 – 517, year 2005, ISSN: 0302-9743, ISBN: 3-540-26208-3.

- F. J. Pelayo, A. Martínez, S. Romero, C. Morillas, E. Ros, E. Fernández, *Cortical Visual Neuroprosthesis for the Blind: Retina-like Software/Hardware Preprocessor*, *1st International IEEE EMBS Conference on Neural Engineering*, Capri (Italia), 20 – 22 March, 2003, ISBN: 0-7803-7579-3.

- F. J. Pelayo, A. Martínez, C. Morillas, S. Romero, L. Sousa, P. Tomás, *Retina-like Processing and Coding Platform for Cortical Neuro-stimulation*, *25th Annual International Conference of the IEEE EMBS*, Cancún (México), 17 – 21 September, 2003, ISBN: 0-7803-7790-7.

- Romero, C. Morillas, A. Martínez, F. J. Pelayo, E. Fernández, *Models and tools for testing visual neuroprosthesis*, *4th Forum of European Neuroscience (FENS)*, Lisboa (Portugal), 2004.

- E. Fernández, C. Morillas, S. Romero, A. Martínez, F. Pelayo, *Neuroengineering Tools for the Design and Test of Visual Neuroprostheses*, congreso *Annual Meeting of The Association for Research in Vision and Ophtalmology*, Fort Lauderdale (Florida), 1 – 5 Mayo 2005, ISSN: 1552-5783, E-Abstract 1483 (publicado en un número electrónico de *Investigative Ophthalmology & Visual Science*, Volume 46, May 2005, http://abstracts.iovs.org/cgi/content/abstract/46/5/1483).

- S. Romero, A. Martínez, Ch. A. Morillas, F. J. Pelayo, E. Fernández, *Visual Information Pre-Processing System for Visual Neuroprostheses*, *XI Congreso Nacional de la Sociedad Española de Neurociencia*. Torremolinos (Málaga). Abstract published in the second volume 41 (2005), number 6 of the *Revista de Neurología*. ISSN: 0210-0010.

- S. Romero, C. Morillas, A. Martínez, F. Pelayo, E. Fernández, *A Research Platform for Visual Neuroprostheses*, *Simposio de Inteligencia Computacional*, *SICO'*2005 (*IEEE Computational Intelligence Society, SC*), págs. 357 – 362, Granada 13 – 16 September, 2005, ISBN: 84-9732-444-7.

- A. Martínez, S. Romero, E. Ros, A. Prieto, F.J. Pelayo, *Implementación de hardware reconfigurable de un modelo de retina*, *II Jornadas sobre Computación Reconfigurable y Aplicaciones* (*JCRA'*2002), pp. 97 – 101, ISBN: 84-699--9448-4, Almuñécar (Granada), 18 – 20 September, 2002.

Apéndice A
Reseña matemática

A.1. La función gaussiana.

A.1.1. La función gaussiana en 1 dimensión.

La función gaussiana en 1 dimensión viene definida de la siguiente forma:

$$f(x;\sigma_x,\mu_x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}[(\frac{x-\mu_x}{\sigma_x})^2]}, \infty < x < \infty \tag{A.1}$$

siendo σ_x la desviación típica y μ_x el valor medio de x.

A.1.2. La función gaussiana bidimensional.

Si las variables x e y no están correladas, la función gaussiana bidimensional es separable, pudiéndose expresar de la siguiente forma:

$$\begin{aligned} f(x,y;\sigma_x,\mu_x,\sigma_y,\mu_y) &= f(x;\sigma_x,\mu_x)\cdot f(y;\sigma_y,\mu_y) = \\ & \frac{1}{2\pi\sigma_x\sigma_y} e^{-\frac{1}{2}[(\frac{x-\mu_x}{\sigma_x})^2+(\frac{y-\mu_x}{\sigma_x})^2]} \\ & \infty < x,y < \infty \end{aligned} \tag{A.2}$$

Si las variables x e y tienen correlación no nula, se la definición se transforma como sigue:

$$\begin{aligned} f(x,y;\sigma_x,\mu_x,\sigma_y,\mu_y) = f(x;\sigma_x,\mu_x)\cdot f(y;\sigma_y,\mu_y) = \\ \frac{1}{2\pi\sigma_x\sigma_y\sqrt{1-\rho^2}} e^{-\frac{1}{2(1-\rho^2)}[(\frac{x-\mu_x}{\sigma_x})^2+(\frac{y-\mu_x}{\sigma_x})^2-2\rho(\frac{x-\mu_x}{\sigma_x})(\frac{y-\mu_y}{\sigma_y})]} \\ \infty < x,y < \infty \end{aligned} \tag{A.3}$$

siendo

$$\rho = \frac{\sigma_x y}{\sigma_x\sigma_y} \tag{A.4}$$

con $\sigma_x y$ el coeficiente de correlación lineal de las variables x e y.
En el desarrollo de la presente tesis doctoral, y habida cuenta del modelo de retina que se ha implementado, ha sido usual encontrarse con gaussianas (y diferencias de gaussianas) donde se han cumplido las condiciones siguientes:

$$\sigma_x = \sigma_y \equiv \sigma \wedge \mu_x = \mu_y = 0 \tag{A.5}$$

con lo que la expresión de la gaussiana bidimensional queda de la siguiente forma:

$$f(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}} \tag{A.6}$$

A.1.3. La función *DoG, Difference of Gaussians* o diferencia de gaussianas.

La función *DoG* viene definida mediante la siguiente expresión:

$$DoG(x, y; \sigma_x, \sigma_y, \mu_x, \mu_y) = f_{gaussiana}(x, \sigma_x, \mu_x) - f_{gaussiana}(y, \sigma_y, \mu_y) \tag{A.7}$$

siendo $f_{gaussiana}$ la función gaussiana bidimensional. Si sendas medias son iguales a cero y $\sigma_x < \sigma_y$ entonces, el trazado de dicha función aproxima la típica forma del sombrero mexicano.

A.1.4. La función sombrero mexicano o *mexican hat*

La función sombrero mexicano se obtiene mediante la normalización de la segunda derivada de la función gaussiana. Es un caso especial de *wavelet* continua. Viene definida mediante la siguiente expresión:

$$\psi(x) = (\frac{2}{\sqrt{3}}\pi^{-\frac{1}{4}})(1 - x^2)e^{\frac{-x^2}{2}} \tag{A.8}$$

A.2. El filtro de *Gabor*

Gabor es un filtro pasobanda en 2 dimensiones definido mediante:

$$g(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} e^{j}(\omega_x x \omega_y y) \tag{A.9}$$

donde σ, ω_x y ω_y son constantes reales y j la unidad imaginaria. De esta forma la gaussiana está modulada mediante los parámetros ω para obtener un filtro direccional.

A.3. El filtro *LoG* o filtro laplaciano de la gaussiana.

El filtro *LoG*, *Laplacian of Gaussian* o laplaciana de la gaussiana está definido mediante la siguiente expresión:

$$LoG(x, y) = \frac{\partial^2 f_{gaussiana}}{\partial x^2} + \frac{\partial^2 f_{gaussiana}}{\partial y^2} = -\frac{1}{\pi\sigma^4}(1 - \frac{x^2 + y^2}{2\sigma^2})e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{A.10}$$

La forma de este filtro es parecida al filtro *DoG* y al sombrero mexicano.

A.4. Convolución.

A.4.1. Integral de convolución en una dimensión.

La convolución entre dos funciones $f(x)$ y $g(y)$ contínuas y definidas en todo el intervalo real está definida por la siguiente ecuación:

$$conv(t) = f \star g = \int_{-\infty}^{+\infty} f(\tau)\, g(t-\tau) d\tau = g \star f \tag{A.11}$$

A.4.2. Integral de convolución en dos dimensiones.

En el caso de dos dimensiones se extiende la definición a:

$$conv_{f,g}(x,y) = f \star g = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(u,v)\, g(x-u, y-v) du\, dv \tag{A.12}$$

A.4.3. Convolución discreta en dos dimensiones.

$$conv_{f,g}(x,y) = f \star g = \sum_m \sum_n f(u,v)\, g(x-m, y-n) \tag{A.13}$$

Apéndice B

Código fuente del módulo de convolución *convolver*

```
-- ======================================================================...  ⇒
...=======
-- CONFIDENTIAL IN CONFIDENCE
-- This confidential and proprietary software may be used
-- only as authorized by a licensing agreement from
-- Politecnico di Torino - Dipartimento di Elettronica
-- or any of its licensee
-- In the event of publication, the following notice is
-- applicable:
--
-- (C) COPYRIGHT 1998 Politecnico di Torino - Dipartimento di Elettron...  ⇒
...ica
--
-- The entire notice above must be reproduced on all authorized copies...  ⇒
....
--
-- Author: Leonardo M. Reyneri
-- Date: 01.01.05
-- Version: 1.4
-- Description: $$$ library of tools CodeSimulink/SMT6040/SMT6041
--
-- $Modification History
-- Date          By              Version         Change description
-- ======================================================================...  ⇒
...=======
--
-- ======================================================================...  ⇒
...=======
-- (c) Politecnico di Torino - Dipartimento di Elettronica
-- ======================================================================...  ⇒
...=======

-- This file contains a CodeSimulink compatible VHDL code which perfor...  ⇒
...ms a 2D
-- convolution over an input matrix
-- Developped by A. Martinez 15/10/04
-- Last revision: 15.06.05
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

library work;
use work.sim_definitions.all;
use work.sim_functions.all;
```

```
use work.sim_protocol_synchpar.all;
use work.sim_fixedpoint_synchpar.all;


--    ////////////
--   // Entity //
--  ////////////
entity sim_convolver_synchpar is --Shift register with parallell outpu...                    ⇒
...t


       -------------
       -- GENERICS --
       -------------

       generic (
                    FIFO_BUFFER_Length: natural;  -- := ...                                  ⇒
...4;
                    BorderValue       : SIM_VECTOR;    ...                                   ⇒
.:=: := (0 => "11111111");
                    DecimColumns      : natural;       ...                                   ⇒
.:=: := 1;
                    DecimRows         : natural;       ...                                   ⇒
...     -- := 1;

-----------------------------------------------------------------
--                  Per la simulazione ParallelSerialMatrix
-----------------------------------------------------------------
-- Implementation : SIM_SIGNAL_ATTRIBUTES := (SST_ParallelSerialMatrix...                    ⇒
..., 40, 0,
--              SSR_unsigned, SOV_wraparound, SRO_round, 0, 5, 5);
-- COEFF : SIM_SIGNAL_ATTRIBUTES := (SST_ParallelSerialMatrix, 40, 0,
--              SSR_unsigned, SOV_wraparound, SRO_round, 0, 5, 5);
-- MULT : SIM_SIGNAL_ATTRIBUTES := (SST_ParallelSerialMatrix, 60, 0,
--              SSR_unsigned, SOV_wraparound, SRO_round, 0, 5, 5);

-------------------------------------------------
--                  Per la simulazione Matrix
-------------------------------------------------
-- Implementation : SIM_SIGNAL_ATTRIBUTES :=
--   (SST_matrix, 8, 0, SSR_unsigned, SOV_wraparound, SRO_round, 0, 5,...                   ⇒
... 5);
-- COEFF : SIM_SIGNAL_ATTRIBUTES :=  (SST_matrix, 8, 0, SSR_unsigned,
--   SOV_wraparound, SRO_round, 0, 5, 5);
-- MULTSIM_SIGNAL_ATTRIBUTES := (SST_matrix, 12, 0, SSR_unsigned,
--              SOV_wraparound, SRO_round, 0, 5, 5);

-----------------------------------------------------------
--                  Per la simulazione ParallelParallel
-----------------------------------------------------------

                    Implementation : SIM_SIGNAL_ATTRIBUTES;
                    -- := (SST_ParallelParallelMatrix, 25*8, 0, SS...                       ⇒
...R_unsigned,
                    --              SOV_wraparound, SRO_round, 0, ...                       ⇒
...5, 5);

                    COEFF : SIM_SIGNAL_ATTRIBUTES;
                    -- :=  (SST_ParallelParallelMatrix, 25*8, 0, S...                       ⇒
...SR_unsigned,
                    --              SOV_wraparound, SRO_round, 0, ...                       ⇒
...5, 5);

                    MULT  : SIM_SIGNAL_ATTRIBUTES;
                    -- :=  (SST_ParallelParallelMatrix, 25*12, 0, ...                       ⇒
...SSR_unsigned,
                    --              SOV_wraparound, SRO_round, 0, ...                       ⇒
...5, 5);
```

```
                    COEFF_VALUES : SIM_VECTOR;
                    -- := (
                    --"00000001", "00000010", "00000011", "0000010...                    ⇒
...0", "00000101",
                    --"00000110", "00000111", "00001000", "0000100...                    ⇒
...1", "00001010",
                    --"00001011", "00001100", "00001101", "0000111...                    ⇒
...0", "00001111",
                    --"00010000", "00010001", "00010010", "0001001...                    ⇒
...1", "00010100",
                    --"00010101", "00010110", "00010111", "0001100...                    ⇒
...0", "00011001");

                    A1 : SIM_SIGNAL_ATTRIBUTES;
                    -- := (SST_matrix, 8, 0, SSR_unsigned, SOV_wra...                    ⇒
...paround,
                    --            SRO_round, 0, 4, 16);
                    Y1                        : SIM_SIGNAL_A...                         ⇒
...TTRIBUTES;
                    -- :=  (SST_matrix, 8, 0, SSR_unsigned, SOV_wr...                    ⇒
...aparound,
                    --            SRO_round, 0, 4, 16);

                    SIM_PIPELINE: INTEGER -- := 1
      );



      ----------
      -- PORTS --
      ----------

      port (

-- PORTE MESSE PER SIMULAZIONE
--      xmaskccnt: out integer range 0 to Implementation.SIM_COLUMNS -...            ⇒
... 1;
--      xINT_DOU : out STD_LOGIC_VECTOR(Implementation.SIM_DATAWIDTH -...            ⇒
... 1 downto 0);
--      xINT_VAL : out SIM_SIGVAL_SYNCHPAR;
--      xINT_RDY : out   STD_LOGIC; -- := '1';
--      xA1_DIN : out   STD_LOGIC_VECTOR(A1.SIM_DATAWIDTH-1 downto 0);
--      xA1_VAL : out   SIM_SIGVAL_SYNCHPAR; -- := (OTHERS => '0');
--      xA1_RDY : in   STD_LOGIC;
--      xINT_DATA : out STD_LOGIC_VECTOR(Implementation.SIM_DATAWIDTH-...            ⇒
...1 downto 0);
--      xMULT_DATA : out STD_LOGIC_VECTOR(MULT.SIM_DATAWIDTH-1 downto ...            ⇒
...0);
--      xMULT_VAL : out SIM_SIGVAL_SYNCHPAR;
--xSUM1_DATA : out
--            STD_LOGIC_VECTOR(Y1.SIM_DATAWIDTH*Implementation.SIM_R...              ⇒
...OWS-1 downto 0);


      A1_DIN      : in STD_LOGIC_VECTOR(A1.SIM_DATAWIDTH - 1 dow...                  ⇒
...nto 0);
      A1_VAL      : in SIM_SIGVAL_SYNCHPAR;
      A1_RDY      : out  STD_LOGIC;

      Y1_DOU      : out STD_LOGIC_VECTOR(Y1.SIM_DATAWIDTH - 1 do...                  ⇒
...wnto 0);
      Y1_VAL      : out SIM_SIGVAL_SYNCHPAR;
      Y1_RDY      : in  STD_LOGIC; -- := '1';

      SIM_CLOCK   : in std_logic;
      SIM_CLEAR   : in std_logic
      );
```

```
end sim_convolver_synchpar;



--    ////////////////
--   // Architecture //
--  ////////////////
architecture sim_struct of sim_convolver_synchpar is


constant InternalDataWidth : integer :=
      Implementation.SIM_DATAWIDTH / num_pes(Implementation);
constant CONVERTER_SIG_ATTR : SIM_SIGNAL_ATTRIBUTES :=
      (A1.SIM_SIGTYPE, InternalDataWidth, Implementation.SIM_BINARYP...           ⇒
...OINT,
      Implementation.SIM_SIGREP, Implementation.SIM_OVERFLOW,
      Implementation.SIM_ROUNDING, A1.SIM_COMPONENTS,
      A1.SIM_ROWS, A1.SIM_COLUMNS);

      CONSTANT SUM1_SIGTYPE : SIM_SIGTYPE  :=
            sim_select(SST_parallelVector,SST_vector,
                  Implementation.SIM_SIGTYPE=SST_matrix);
      CONSTANT SUM1_DATAWIDTH : INTEGER :=
            sim_select(Y1.SIM_DATAWIDTH*Implementation.SIM_ROWS,Y1...            ⇒
....SIM_DATAWIDTH,
            Implementation.SIM_SIGTYPE=SST_matrix);
      CONSTANT SUM1 : SIM_SIGNAL_ATTRIBUTES :=
            (SUM1_SIGTYPE, SUM1_DATAWIDTH, Y1.SIM_BINARYPOINT, Y1....            ⇒
...SIM_SIGREP,
            Y1.SIM_OVERFLOW, Y1.SIM_ROUNDING, Implementation.SIM_R...            ⇒
...OWS, 0, 0);
      CONSTANT SUM2           : SIM_SIGNAL_ATTRIBUTES := (SS...                    ⇒
...T_scalar,
            Y1.SIM_DATAWIDTH, Y1.SIM_BINARYPOINT, Y1.SIM_SIGREP,
            Y1.SIM_OVERFLOW, Y1.SIM_ROUNDING, 0, 0, 0);
      CONSTANT LATENCY        : INTEGER :=
            A1.SIM_COLUMNS * ((Implementation.SIM_ROWS)/2) +
                  ((Implementation.SIM_COLUMNS)/2) + 1;

      SIGNAL INT_DATA   : STD_LOGIC_VECTOR(Implementation.SIM_...                ⇒
...DATAWIDTH-1 downto 0);
      SIGNAL INT_VALSIM_SIGVAL_SYNCHPAR;
      SIGNAL INT_RDYSTD_LOGIC;
      SIGNAL COEFF_DATA : STD_LOGIC_VECTOR(COEFF.SIM_DATAWIDTH-1 dow...          ⇒
...nto 0);
      SIGNAL COEFF_VAL  : SIM_SIGVAL_SYNCHPAR;
      SIGNAL COEFF_RDY  : STD_LOGIC;
      SIGNAL MULT_DATA  : STD_LOGIC_VECTOR(MULT.SIM_DATAWIDTH-...                ⇒
...1 downto 0);
      SIGNAL MULT_VAL   : SIM_SIGVAL_SYNCHPAR;
      SIGNAL MULT_RDY   : STD_LOGIC;
      SIGNAL SUM1_DATA  : STD_LOGIC_VECTOR(SUM1.SIM_DATAWIDTH-...                ⇒
...1 downto 0);
      SIGNAL SUM1_VAL   : SIM_SIGVAL_SYNCHPAR;
      SIGNAL SUM2_VAL   : SIM_SIGVAL_SYNCHPAR;
      SIGNAL SUM1_RDY   : STD_LOGIC;  -- Se usa mult_rdy
      SIGNAL SUM2_RDY   : STD_LOGIC;


type SHIFT_SIGNAL_TYPE is array (0 to Implementation.SIM_ROWS - 1) of
      std_logic_vector (InternalDataWidth - 1 downto 0);


--------------------
-- Custom signals --
```

```
-------------------

signal A1_DINC                              : std_logic_ve...                    ⇒
...ctor (InternalDataWidth - 1
                                                        ...                      ⇒
...          downto 0);

-- <FIFO_BUFFER SIGNALS> --
signal FIFO_BUFFER_Out        : std_logic_vector (InternalDa...                  ⇒
...taWidth - 1
                                                        ...                      ⇒
...          downto 0);
signal FIFO_BUFFER_EIS        : std_logic; -- 21_OCT
signal FIFO_BUFFER_Not_Full   : std_logic;
signal FIFO_BUFFER_Empty      : std_logic;
signal FIFO_BUFFER_NOT_Empty  : std_logic;
signal FIFO_BUFFER_READ       : std_logic;
-- </FIFO_BUFFER SIGNALS> --

signal SHIFT_SIGNAL                 : SHIFT_SIGNAL_TYPE;
--signal SHIFT_SIGNAL_EIS          : std_logic;

-- Output registers
type RegsTYPE is array (0 to Implementation.SIM_ROWS - 1,
           0 to Implementation.SIM_COLUMNS - 1) of
                std_logic_vector (InternalDataWidth - 1 downto...               ⇒
... 0);

signal Regs                                 : Regs...                            ⇒
...TYPE;


---------------------
-- Protocol signals --
---------------------

-- From CodeSimulink docs: when high, output register of associated
-- cell shall be updated at next rising edge of clock
-- input data might not remain valid afterwards!
signal P_EN : std_logic;
signal P_EN2: std_logic;

-- Four bits with information about: VAL, EOV, EOM, GRD
signal val : SIM_SIGVAL_SYNCHPAR_VECTOR(0 to 0);
--signal valid : std_logic;

-- e_is: when high, indicates that the whole input data (either scalar...          ⇒
..., --
-- vector matrix) has been completely received at next rising edge of ...          ⇒
...SIM_CLOCK
-- sometimes should be anded with NOMOREVALID to be sure that input da...          ⇒
...ta
-- won't be valid after next clock


-- indicates that a new data can be sent to output;
signal nextdata   : std_logic;
signal e_is, e_is2 : std_logic;

--signal receiving                  : std_logic;
--signal FIFO_BUFFER_RECEIVING      : std_logic;
--signal IN_RDY : std_logic;

signal MaskRCnt              : integer range 0 to Implement...                   ⇒
...ation.SIM_ROWS - 1;
signal MaskCCnt              : integer range 0 to Implement...                   ⇒
...ation.SIM_COLUMNS - 1;
```

```
-- Contador de entrada
signal Rows_Cnt          : integer range 0 to A1.SIM_ROWS - 1;
signal Columns_Cnt       : integer range 0 to A1.SIM_COLUMNS - ...                    ⇒
...1;


--signal Int_Output            : std_logic_vector (
      -- InternalDataWidth * num_pes(Implementation) - 1 downto 0);
--signal Ext_Output            : std_logic_vector (
      -- InternalDataWidth * num_pes(Implementation) - 1 downto 0);
signal Int_Output        : std_logic_vector (Implementation.SIM...                    ⇒
..._DATAWIDTH - 1 downto 0);
signal Ext_Output        : std_logic_vector (Implementation.SIM...                    ⇒
..._DATAWIDTH - 1 downto 0);
signal out_en            : std_logic;
signal sreg_receiving, sreg_ei$ std_logic;



begin


--      ------------------------------------
--      -- <Temporal input for simulation> --
--      ------------------------------------
--      entrada : sim_constant_synchPar
--      GENERIC MAP (
--              Y1      => A1,
--              SIM_PIPELINE => 1,
--              CONST_DATA => (
--              "00000000", "00000001", "00000010", "00000011", "00000...            ⇒
...100", "00000101", "00000110", "00000111",
--              "00001000", "00001001", "00001010", "00001011", "00001...            ⇒
...100", "00001101", "00001110", "00001111",
--              "00010000", "00010001", "00010010", "00010011", "00010...            ⇒
...100", "00010101", "00010110", "00010111",
--              "00011000", "00011001", "00011010", "00011011", "00011...            ⇒
...100", "00011101", "00011110", "00011111",
--              "00100000", "00100001", "00100010", "00100011", "00100...            ⇒
...100", "00100101", "00100110", "00100111",
--              "00101000", "00101001", "00101010", "00101011", "00101...            ⇒
...100", "00101101", "00101110", "00101111",
--              "00110000", "00110001", "00110010", "00110011", "00110...            ⇒
...100", "00110101", "00110110", "00110111",
--              "00111000", "00111001", "00111010", "00111011", "00111...            ⇒
...100", "00111101", "00111110", "00111111")
--              -- 0..63, 64 valores.
--      )
--      PORT MAP (
--              Y1_DOUT => A1_DIN,
--              Y1_VAL => A1_VAL,
--              Y1_RDY => xxA1_RDY,
--              SIM_CLEAR    => SIM_CLEAR,
--              SIM_CLOCK    => SIM_CLOCK
--      );
--      xA1_DIN <= A1_DIN;
--      xA1_VAL <= A1_VAL;
--      xxA1_RDY <= xA1_RDY and A1_RDY;
--      -------------------------------------
--      -- </Temporal input for simulation> --
--      -------------------------------------




      ------------------
      -- <PROTOCOL_1> --
      ------------------
```

```
      val(0) <= A1_VAL;
      prot1:sim_prot_synchpar
            generic map(
                  SIM_PIPELINE => 1,
                  SIM_INPUTS_NUMBER => 1,
                  A                       => A1,
                  Y1                      => A1
            )
            port map(
                  IN_VAL_VECTOR => val,
                  --VALID        => valid, -- when high, indi...                    ⇒
...cates that input
                        --data is valid, parecido a p_en, se m...                  ⇒
...antiene.
                  IN_RDY(0)     => A1_RDY, -- vector of input...                   ⇒
... ready signals;
                  P_EN          => P_EN, -- the whole input da...                  ⇒
...ta has been completely
                        --received, (es un pulso)
                  READY         => FIFO_BUFFER_Not_Full,
                  E_IS          => E_IS,
--                RECEIVING     => RECEIVING,
                  SIM_CLOCK     => SIM_CLOCK,
                  SIM_CLEAR     => SIM_CLEAR
            );
      ------------------
      -- </PROTOCOL_1> --
      ------------------


      -- First of All performs a conversion with the input data (Inp...             ⇒
...ut Data Converter)
      idc: sim_convert_synchpar generic map (A1, CONVERTER_SIG_ATTR)...           ⇒
... port map (A1_DIN, A1_DINC);




      -----------------
      -- <INPUT FIFO> --
      -----------------
blfi: block
      -- 2 bit more per saving End-of-Image (signal e_is from CS's p...           ⇒
...rotocol) and receiving
--    type FIFO_BUFFER_Type is array (0 to FIFO_BUFFER_Length - 1) o...           ⇒
...f std_logic_vector (InternalDataWidth + 1 downto 0);
      type FIFO_BUFFER_Type is array (0 to FIFO_BUFFER_Length - 1) o...           ⇒
...f std_logic_vector (InternalDataWidth downto 0);
      type FIFO_BUFFER_SM_Type is (FSM_Ini, FSM_Working);

      signal FIFO_BUFFER_cntIn, FIFO_BUFFER_cntOut INTEGER RANG...                ⇒
...E 0 TO FIFO_BUFFER_Length - 1;
      signal Fifo_BUFFER                              ...                         ⇒
...           : FIFO_BUFFER_TYPE;
      signal FIFO_BUFFER_FSM                          ...                         ⇒
...     : FIFO_BUFFER_SM_Type;

      begin

      -- Write pointer (FIFO_IN Input)
      process (SIM_CLOCK)
      begin
            if SIM_CLOCK'Event and SIM_CLOCK = '1' then
                  if SIM_CLEAR = '1' then
                        FIFO_BUFFER_cntIn <= 0;
                  elsif P_EN = '1' and FIFO_BUFFER_Not_Full = '1...               ⇒
...' then -- all inputs received
```

```
                        if FIFO_BUFFER_cntIn = FIFO_BUFFER_Len...                    ⇒
...gth - 1 THEN
                              FIFO_BUFFER_cntIn <= 0;
                        else
                              FIFO_BUFFER_cntIn <= FIFO_BUFF...                      ⇒
...ER_cntIn + 1;
                        end if;
                  end if;
            END IF;
      end process;


      -- Read pointer (FIFO_IN Output)
      process (SIM_CLOCK)
      begin
            if SIM_CLOCK'Event and SIM_CLOCK = '1' then
                  if SIM_CLEAR = '1' then
                        FIFO_BUFFER_cntOut <= 0;
                  elsif FIFO_BUFFER_READ = '1' then -- TODO
                        if FIFO_BUFFER_cntOut = A1.SIM_COLUMNS...                    ⇒
...-1 THEN
                              FIFO_BUFFER_cntOut <= 0;
                        else
                              FIFO_BUFFER_cntOut <= FIFO_BUF...                      ⇒
...FER_cntOut + 1;
                        end if;
                  end if;
            END IF;
      end process;


      FIFO_BUFFER_Empty <= '1' when FIFO_BUFFER_cntIn = FIFO_BUFFER_...             ⇒
...cntOut AND FIFO_BUFFER_FSM = FSM_Ini else '0';
      FIFO_BUFFER_NOT_Empty <= not FIFO_BUFFER_Empty;
      FIFO_BUFFER_Not_Full <= '0' when FIFO_BUFFER_cntIn = FIFO_BUFF...             ⇒
...ER_cntOut AND FIFO_BUFFER_FSM = FSM_Working else '1';

      process (SIM_CLOCK)
      begin
            if SIM_CLOCK'Event and SIM_CLOCK = '1' then
                  if SIM_CLEAR = '1' then
                        FIFO_BUFFER_FSM <= FSM_Ini;
                  else
                        case FIFO_BUFFER_FSM is
                              when FSM_Ini =>
                                    if (P_EN = '1' AND FIF...                        ⇒
...O_BUFFER_READ = '0') then
                                          FIFO_BUFFER_FS...                          ⇒
...M <= FSM_Working;
                                    end if;
                              when FSM_Working =>
                                    if (P_EN = '0' AND FIF...                        ⇒
...O_BUFFER_READ = '1') then
                                          FIFO_BUFFER_FS...                          ⇒
...M <= FSM_Ini;
                                    end if;
                        end case;
                  end if;
            end if;
      end process;

      -- Performs the scroll over the FIFO
      process (SIM_CLOCK)
      begin
            if SIM_CLOCK'Event and SIM_CLOCK = '1' then
                  if P_EN = '1' then
--                      Fifo_BUFFER(FIFO_BUFFER_cntIn) <= RECE...                  ⇒
...IVING & e_is & A1_DINC;
```

```
                        Fifo_BUFFER(FIFO_BUFFER_cntIn) <= e_is...                  ⇒
... & A1_DINC;
                end if;
            end if;
      end process;

      -- Output
      FIFO_BUFFER_Out <= Fifo_BUFFER (FIFO_BUFFER_cntOut) (InternalD...            ⇒
...ataWidth - 1
            downto 0);
      FIFO_BUFFER_EIS <= Fifo_BUFFER (FIFO_BUFFER_cntOut) (InternalD...            ⇒
...ataWidth);
      --      FIFO_BUFFER_RECEIVING <= Fifo_BUFFER (FIFO_BUFFER_cn...              ⇒
...tOut)
      --      (InternalDataWidth + 1);

      end block;

      ------------------
      -- </INPUT FIFO> --
      ------------------




      ----------------------
      -- <SHIFT REGISTERS> --
      ----------------------
      blfo: block
            type SHIFTREG_Type is array (0 to A1.SIM_COLUMNS - 1) ...              ⇒
...of
                  std_logic_vector ((Implementation.SIM_ROWS - 1...                ⇒
...)*InternalDataWidth - 1 downto 0);
            type ShiftReg_SM_Type is (FSM_Ini, FSM_Working);

            signal ShiftReg         : SHIFTREG_Type;
            signal ShiftReg_Joker : std_logic_vector (
                  (Implementation.SIM_ROWS - 1)*InternalDataWidt...                ⇒
...h - 1 downto 0);
            signal ShiftReg_Joker2 : std_logic_vector (
                  (Implementation.SIM_ROWS - 1)*InternalDataWidt...                ⇒
...h - 1 downto 0);
--          signal ShiftReg_FULL         : std_logic;
            signal ShiftReg_FSM     : ShiftReg_SM_Type;
            signal cntShift, cntShift_norey integer rang...                        ⇒
...e 0 to A1.SIM_COLUMNS - 1;
            signal cntShiftOut : integer range 0 to A1.SIM_COLUMNS...              ⇒
... * (Implementation.SIM_ROWS - 1) + (Implementation.SIM_COLUMNS) / 2 - ...       ⇒
...1;


            begin

            -- Biestable SR, Salida sreg_receiving
            process (SIM_CLOCK)
            begin
                  if (SIM_CLOCK'Event) and (SIM_CLOCK = '1') the...                ⇒
...n
                        if SIM_CLEAR = '1' then
                              sreg_receiving <= '0';
                        elsif (FIFO_BUFFER_EIS = '1') then
                                    sreg_receiving <= '0';...                      ⇒
...
                        elsif FIFO_BUFFER_EMPTY = '0' then -- ...                  ⇒
...si ce' eis c'e empty
                                    sreg_receiving <= '1';...                      ⇒
...
```

```
                  end if;
            end if;
      end process;


      -- Write pointer (FIFO Input)
      process (SIM_CLOCK, SIM_CLEAR, FIFO_BUFFER_Empty,
            cntShift, FIFO_BUFFER_READ, sreg_receiving)
      begin

            if SIM_CLEAR = '1' then
                  cntShift_noreg <= 0;

            elsif (FIFO_BUFFER_READ = '1') then
                  if cntShift = A1.SIM_COLUMNS - 1  THEN...                    ⇒
...
                        cntShift_noreg <= 0;
                  else
                        cntShift_noreg <= cntShift + 1...                      ⇒
...;
                  end if;
            else
                  cntShift_noreg <= cntShift;
            end if;
            if SIM_CLOCK'Event and SIM_CLOCK = '1' then
                  -- Controls the End of Image e_is
                  cntShift <= cntShift_noreg;
            end if;
      end process;


      -- OutPut counter
      process (SIM_CLOCK)
      begin
            if SIM_CLOCK'Event and SIM_CLOCK = '1' then
                  if FIFO_BUFFER_Empty = '0' then
                        if SIM_CLEAR = '1' then
                              cntShiftOut <= 0;
                        elsif FIFO_BUFFER_READ = '1' t...                      ⇒
...hen
                              if cntShiftOut = A1.SI...                        ⇒
...M_COLUMNS * A1.SIM_ROWS +
                                                A1.SIM...                      ⇒
..._COLUMNS * (Implementation.SIM_ROWS - 1) +
                                                (Imple...                      ⇒
...mentation.SIM_COLUMNS) / 2 - 1
                                          then
                                          cntShiftOut <=...                    ⇒
... 0;
                              else
                                    cntShiftOut <=...                          ⇒
... cntShiftOut + 1;
                              end if;
                        end if;
                  end if;
            end if;
      end process;

      -- Performs the scroll
      process (SIM_CLOCK)
      begin
            if SIM_CLOCK'Event and SIM_CLOCK = '1' then
                        ShiftReg_Joker2 <= ShiftReg(cn...                      ⇒
...tShift_noreg);
            end if;
      end process;

      ShiftReg_Joker <= ShiftReg_Joker2(
```

```
                  (Implementation.SIM_ROWS - 2)*InternalDataWidt...                ⇒
...h - 1 DOWNTO 0) &
                        FIFO_BUFFER_Out (InternalDataWidth - 1...                  ⇒
... DOWNTO 0);
            process (SIM_CLOCK)
            begin
                  if SIM_CLOCK'Event and SIM_CLOCK = '1' then
                        if FIFO_BUFFER_READ = '1' then  -- Il ...                  ⇒
...nostro   RDY
                              ShiftReg(cntShift) <= ShiftReg...                    ⇒
...Joker;
                        end if;
                  end if;
            end process;


            connect: for i in 0 to Implementation.SIM_ROWS - 1 gen...              ⇒
...erate
                  ifg1: if i = 0 generate
                        SHIFT_SIGNAL (0) <= FIFO_BUFFER_Out (
                              InternalDataWidth - 1 DOWNTO 0...                    ⇒
...);
                  end generate;
                  ifg2: if i > 0 generate
                        SHIFT_SIGNAL (i) <= ShiftReg_Joker2 ((...                  ⇒
...i-1) * InternalDataWidth +
                              InternalDataWidth - 1 downto (...                    ⇒
...i-1) * InternalDataWidth);
                  end generate;
            end generate;

--          xSHIFT_SIGNAL0 <= SHIFT_SIGNAL(0);
--          xSHIFT_SIGNAL1 <= SHIFT_SIGNAL(1);
--          xSHIFT_SIGNAL2 <= SHIFT_SIGNAL(2);
--          xSHIFT_SIGNAL3 <= SHIFT_SIGNAL(3);
--          xSHIFT_SIGNAL4 <= SHIFT_SIGNAL(4);

      end block;
      -----------------------
      -- </SHIFT REGISTERS> --
      -----------------------


      -----------------------
      -- <OUTPUT REGISTERS> --
      -----------------------
      process (SIM_CLOCK)
      begin
            if SIM_CLOCK'Event and SIM_CLOCK = '1' then
                  if FIFO_BUFFER_READ = '1' then
                        for j in 0 to Implementation.SIM_ROWS ...                  ⇒
...- 1 loop
                              Regs(j,0) <= SHIFT_SIGNAL (j);...                    ⇒
...
                              for i in 1 to Implementation.S...                    ⇒
...IM_COLUMNS - 1 loop
                                    Regs(j,i) <= Regs (j,i...                      ⇒
...-1);                                          ...                               ⇒
...
                              end loop;
                        end loop;
                  end if;
            end if;
      end process;
      -----------------------
      -- </OUTPUT REGISTERS> --
      -----------------------
```

```
--      xregs0 <= regs(0,0);
--      xregs1 <= regs(0,1);
--      xregs5 <= regs(1,0);
--      xregs24 <= regs(4,4);

      ------------------
      -- <PROTOCOL_2> --
      ------------------

-- REPLICA signal Rows_Cnt_Joker     : integer range - (Implementat...                    ⇒
...ion.SIM_ROWS/2) to 0;
-- REPLICA signal Columns_Cnt_Joker : integer range - (Implementat...                    ⇒
...ion.SIM_COLUMNS/2) - 1 to 0;
-- REPLICA signal Rows_Cnt                 : integer range 0 to A...                    ⇒
...1.SIM_ROWS - 1;
-- REPLICA signal Columns_Cnt       : integer range 0 to A1.SIM_CO...                    ⇒
...LUMNS - 1;


      b_out_en: block is
      signal x: std_logic;
      signal latency_cnt: integer range 0 to LATENCY;

            begin
            -- REPLICA signal latency_cnt : integer range 0 to L...                     ⇒
...ATENCY - 1;
            process (SIM_CLOCK, SIM_CLEAR)
            begin
                  if SIM_CLOCK'Event and SIM_CLOCK = '1' then
                        if SIM_CLEAR = '1' or sreg_receiving =...                      ⇒
... '0' then
                              latency_cnt <= 0;
                        elsif FIFO_BUFFER_READ = '1' then
                              if latency_cnt /= LATENCY then...                        ⇒
...
                                    latency_cnt <= latency...                          ⇒
...._cnt + 1;
                              end if;
                        end if;
                  end if;
            end process;


            x <= '1' when (latency_cnt = (LATENCY-1)) and (FIFO_BU...                  ⇒
...FFER_READ = '1') else '0';
            -- Biestable SR
            process (SIM_CLOCK)
            begin
                  if SIM_CLOCK'Event and SIM_CLOCK='1' then
                        if SIM_CLEAR = '1' then
                              out_en <= '0';
                        elsif x = '1' then
                              out_en <= '1';
                        elsif sreg_eis = '1' then
                              out_en <= '0';
                        end if;
                  end if;
            end process;
--          -- Nota: Para que no se retrase la salida de out_en_jo...                  ⇒
...ker. Variable comodin...
--          --out_en <= (out_en_joker or x) and (not y);
--          out_en <= (out_en_joker or x) and (not sreg_eis);

            process (SIM_CLOCK, SIM_CLEAR)
            begin
                  if SIM_CLOCK'Event and SIM_CLOCK = '1' then
                        if (SIM_CLEAR = '1') or (out_en = '0')...                      ⇒
... then
```

```
                        Columns_Cnt <= 0;
                    elsif FIFO_BUFFER_READ = '1' then
                        if Columns_Cnt = A1.SIM_Column...                    ⇒
...s - 1 then
                            Columns_Cnt <= 0;
                        else
                            Columns_Cnt <= Columns...                        ⇒
..._Cnt + 1;
                        end if;
                    end if;
                end if;
            end process;

            process (SIM_CLOCK, SIM_CLEAR)
            begin
                if SIM_CLOCK'Event and SIM_CLOCK = '1' then
                    if SIM_CLEAR = '1' or (out_en = '0') t...                ⇒
...hen
                        Rows_Cnt <= 0;
                    elsif (FIFO_BUFFER_READ = '1') and (Co...                ⇒
...lumns_Cnt = A1.SIM_Columns - 1) then
                        if Rows_Cnt = A1.SIM_Rows - 1 ...                    ⇒
...then
                            Rows_Cnt <= 0;
                        else
                            Rows_Cnt <= Rows_Cnt +...                        ⇒
... 1;
                        end if;
                    end if;
                end if;
            end process;

            sreg_eis <= '1' when ((Columns_Cnt = A1.SIM_Columns - ...         ⇒
...1) and (Rows_Cnt = A1.SIM_Rows - 1))
                                        and (nextdata ...                    ⇒
...= '1') and (e_is2 = '1') else '0';
            end block;

        prot2: sim_protgen_synchpar
            generic map(
                SIM_PIPELINE => SIM_PIPELINE,
                Y1                      => Implementat...                    ⇒
...ion
            )
            port map(
                IN_EN       => out_en, --when high indicat...                ⇒
...es that data for output is valid at least until next rising edge of cl...  ⇒
...ock
--              IN_RDY      => IN_RDY, -- when high indica...                ⇒
...tes that the associated cell is allowed to send new data
                P_EN        => P_EN2, -- the whole input d...                ⇒
...ata has been completely received, -- es un pulso
                Y1_VAL      => INT_VAL,
                Y1_RDY      => INT_RDY,
                NEXT_OUTDATA=> nextdata,
--              Y1_RDY      => FIFO_BUFFER_Not_Full,
                e_is        => e_is2,
                CNTR        => MaskRCnt,
                CNTC        => MaskCCnt,
                SIM_CLOCK   => SIM_CLOCK,
                SIM_CLEAR   => SIM_CLEAR
            );

--          xmaskccnt <= maskccnt;
        ------------------
        -- </PROTOCOL_2> --
        ------------------
```

```
      -------------------------
      -- <CALCULATING OUTPUT> --
      -------------------------



                  FIFO_BUFFER_READ <= e_is2 and nextdata and FIF...                  ⇒
...O_BUFFER_Not_Empty when sreg_receiving = '1' and out_en = '1' -- c'e' ...         ⇒
...un'immagine in ingresso ed una in uscita
                        else FIFO_BUFFER_Not_Empty when sreg_r...                    ⇒
...eceiving = '1' and out_en = '0' -- c'e' un'immgine in ingresso ma non ...         ⇒
...in uscita
                        else e_is2 and nextdata when sreg_rece...                    ⇒
...iving = '0' and out_en = '1' -- c'e' un'immagine in uscita ma non in i...         ⇒
...ngresso
                        else '0';

      --------------------
      -- Parallel-Serial--
      --------------------

      imp1: if Implementation.SIM_SIGTYPE = SST_ParallelSerialMatrix...              ⇒
... generate
            b1: block is
            begin

                  -- Output from this block
                  process (MaskRCnt, MaskCCnt, Rows_Cnt, Columns...                  ⇒
...._Cnt, Regs)
                  variable row : INTEGER RANGE - Implementation....                  ⇒
...SIM_ROWS + 1 TO A1.SIM_ROWS - 1;
                  variable col : INTEGER RANGE - Implementation....                  ⇒
...SIM_COLUMNS + 1 TO A1.SIM_COLUMNS - 1;
                  begin
                        col := Columns_Cnt - MaskCCnt;
                        FOR j IN 0 TO Implementation.SIM_ROWS ...                    ⇒
...- 1 loop
                              row := Rows_Cnt - j;
                              if
                                    (Row >= - (Implementat...                        ⇒
...ion.SIM_ROWS/2))  and
                                    (Row < A1.SIM_ROWS - I...                        ⇒
...mplementation.SIM_ROWS/2) and
                                    (Col >= - (Implementat...                        ⇒
...ion.SIM_COLUMNS/2))  and
                                    (Col < A1.SIM_COLUMNS ...                        ⇒
...- Implementation.SIM_COLUMNS/2) then
                                          Int_Output (j ...                          ⇒
...* InternalDataWidth + InternalDataWidth - 1 downto j * InternalDataWid...       ⇒
...th) <= Regs (j,MaskCCnt); -- HOT: specular image
                              else -- BorderValue is bigendi...                      ⇒
...an
                                    for k in 0 to Internal...                        ⇒
...DataWidth - 1 loop
                                          Int_Output (j ...                          ⇒
...* InternalDataWidth + InternalDataWidth - 1 - k) <= BorderValue(0, k);...       ⇒
...
                                    end loop;
                              end if;
                        end loop;
                  end process;
            end block;

      end generate;
```

```
      ------------------------------------------------------------...                    ⇒
...
      -- Matrix (Serial-Serial MaskR * MaskC clock cycles needed) --...                  ⇒
...
      ------------------------------------------------------------...                    ⇒
...
      imp2: if Implementation.SIM_SIGTYPE = SST_matrix generate
            b1: block is

            begin

                  process (MaskCCnt, MaskRCnt, Regs, Rows_Cnt, C...                     ⇒
...olumns_Cnt)
                  variable row : INTEGER RANGE - Implementation....                     ⇒
...SIM_ROWS + 1 TO A1.SIM_ROWS - 1;
                  variable col : INTEGER RANGE - Implementation....                     ⇒
...SIM_COLUMNS + 1 TO A1.SIM_COLUMNS - 1;
                  begin
--                      FOR j IN 0 TO Implementation.SIM_ROWS ...                     ⇒
...- 1 loop
--                            FOR i IN 0 TO Implementation.S...                       ⇒
...IM_COLUMNS - 1 loop
                                    row := Rows_Cnt - Mask...                       ⇒
...RCnt;
                                    col := Columns_Cnt - M...                       ⇒
...askCCnt;
                                    if
                                          (Row >= - Impl...                         ⇒
...ementation.SIM_ROWS/2)  and
                                          (Row < A1.SIM_...                         ⇒
...ROWS - Implementation.SIM_ROWS/2) and
                                          (Col >= - Impl...                         ⇒
...ementation.SIM_COLUMNS/2)  and
                                          (Col < A1.SIM_...                         ⇒
...COLUMNS - Implementation.SIM_COLUMNS/2) then
                                          Int_Output (Im...                         ⇒
...plementation.SIM_DATAWIDTH - 1 downto 0) <= Regs (MaskRCnt,MaskCCnt); ...        ⇒
...
                                                --</AM...                          ⇒
...A HOT>
                                    else -- BorderValue is...                       ⇒
... bigendian
                                          for k in 0 to ...                         ⇒
...InternalDataWidth - 1 loop
                                                --<AMA...                          ⇒
... HOT>
                                                --Int_...                          ⇒
...Output ((i*Implementation.SIM_COLUMNS+j) * InternalDataWidth + Interna...        ⇒
...lDataWidth - 1 - k) <= BorderValue(0, k);
--                                              Int_Ou...                          ⇒
...tput (Implementation.SIM_DATAWIDTH - 1 downto k) <= BorderValue(0,k); ...        ⇒
...-- "11111111"; -- HELP --BorderValue(0, k); -- HOT
                                          Int_Output (In...                         ⇒
...ternalDataWidth - 1 - k) <= BorderValue(0, k);
                                                --</AM...                          ⇒
...A HOT>
                                          end loop;
                                    end if;
--                            end loop;
--                      end loop;
                  end process;
            end block;
      end generate;


      ----------------------
      -- Parallel-Parallel --
```

```
      ----------------------

      imp3: if Implementation.SIM_SIGTYPE = SST_ParallelParallelMatr...           ⇒
...ix generate
            b1: block is

            begin

            -- Output from this block
                  process (MaskCCnt, MaskRCnt, Regs, Rows_Cnt, C...              ⇒
...olumns_Cnt)
                  variable row : INTEGER RANGE - Implementation....              ⇒
...SIM_ROWS + 1 TO A1.SIM_ROWS - 1;
                  variable col : INTEGER RANGE - Implementation....              ⇒
...SIM_COLUMNS + 1 TO A1.SIM_COLUMNS - 1;
                  begin
                        FOR j IN 0 TO Implementation.SIM_ROWS ...                ⇒
...- 1 loop
                              FOR i IN 0 TO Implementation.S...                  ⇒
...IM_COLUMNS - 1 loop
                                    Row := Rows_Cnt - j;
                                    Col := Columns_Cnt - i...                    ⇒
...;
                                    if
                                          (Row >= - Impl...                      ⇒
...ementation.SIM_ROWS/2)  and
                                          (Row < A1.SIM_...                      ⇒
...ROWS - Implementation.SIM_ROWS/2) and
                                          (Col >= - Impl...                      ⇒
...ementation.SIM_COLUMNS/2)  and
                                          (Col < A1.SIM_...                      ⇒
...COLUMNS - Implementation.SIM_COLUMNS/2) then

                                          Int_Output ((j...                      ⇒
...*Implementation.SIM_COLUMNS+i) * InternalDataWidth + InternalDataWidth...     ⇒
... - 1 downto
                                                   ...                           ⇒
..(j*Implementation.SIM_COLUMNS+i) * InternalDataWidth) <= Regs (j,i);...        ⇒
...
                                    else -- BorderValue is...                    ⇒
... bigendian
                                          for k in 0 to ...                      ⇒
...InternalDataWidth - 1 loop
                                                Int_Ou...                        ⇒
...tput ((i*Implementation.SIM_COLUMNS+j) * InternalDataWidth + InternalD...     ⇒
...ataWidth - 1 - k) <= BorderValue(0, k);
                                          end loop;
--                                              Int_Ou...                        ⇒
...tput ((j*Implementation.SIM_COLUMNS+i) * InternalDataWidth + InternalD...     ⇒
...ataWidth - 1 downto
--    ???                                          ...                          ⇒
. (j*Implementation.SIM_COLUMNS+i) * InternalDataWidth) <= "11111111";...        ⇒
...
                                    end if;
                              end loop;
                        end loop;
                  end process;
            end block;

      end generate;




      sim_out_register(SIM_CLEAR, SIM_CLOCK, P_EN2, SIM_PIPELINE, In...          ⇒
...t_Output, Ext_Output, sim_implement_synchPar);

      INT_DATA <= Ext_Output;
```

```
--      xINT_DATA <= INT_DATA;

        --------------------------
        -- </CALCULATING OUTPUT> --
        --------------------------



        -------------
        -- CONSTANT --
        -------------
        I_sim_constant1 : sim_constant_synchPar
        GENERIC MAP (
              Y1    => COEFF,
              SIM_PIPELINE=> 0,
              CONST_DATA  => COEFF_VALUES
        )
        PORT MAP (
              Y1_DOU=> COEFF_DATA,
              Y1_VAL=> COEFF_VAL,
              Y1_RDY=> COEFF_RDY,
              SIM_CLEAR   => SIM_CLEAR,
              SIM_CLOCK   => SIM_CLOCK
        );


        ---------------
        -- MULTIPLIER --
        ---------------
        pr : sim_product2_synchPar
        GENERIC MAP (
              Y1    => MULT,
              A1    => Implementation,
              A2    => COEFF,
              SIM_PIPELINE=> 1,
              SIM_DIRECTION=> "* *"
        )
        PORT MAP (
              Y1_DOU=> MULT_DATA,
              Y1_VAL=> MULT_VAL,
              Y1_RDY=> MULT_RDY,
              A1_DIN=> INT_DATA,
              A1_VAL=> INT_VAL,
              A1_RDY=> INT_RDY,
              A2_DIN=> COEFF_DATA,
              A2_VAL=> COEFF_VAL,
              A2_RDY=> COEFF_RDY,
              SIM_CLEAR   => SIM_CLEAR,
              SIM_CLOCK   => SIM_CLOCK
        );

--      xMULT_DATA <= MULT_DATA; -- per simulazione
--      xMULT_VAL <= MULT_VAL; -- per simulazione


        ----------
        -- SUM1 --
        ----------
        s1 : sim_sum1_synchPar
        GENERIC MAP (
              Y1    => SUM1,
              A1    => MULT,
              SIM_PIPELINE=> 1
        )
        PORT MAP (
              Y1_DOU=> SUM1_DATA,
              Y1_VAL=> SUM1_VAL,
              Y1_RDY=> SUM1_RDY,      -- DEBUG: RDY FINAL
```

```
            A1_DIN=> MULT_DATA,
            A1_VAL=> MULT_VAL,
            A1_RDY=> MULT_RDY,

            SIM_CLEAR   => SIM_CLEAR,
            SIM_CLOCK   => SIM_CLOCK
      );

      --xSUM1_DATA <= SUM1_DATA;



      ---------
      -- SUM2 --
      ---------
      s2 : sim_sum1_synchPar
      GENERIC MAP (
            Y1    => SUM2,
            A1    => SUM1,
            SIM_PIPELINE=> 1
      )
      PORT MAP (
            Y1_DOU=> Y1_DOU,        -- Uscita finale
            Y1_VAL=> SUM2_VAL,      -- Uscita finale
            Y1_RDY=> SUM2_RDY,      -- Ingresso alla fine

            A1_DIN=> SUM1_DATA,
            A1_VAL=> SUM1_VAL,
            A1_RDY=> SUM1_RDY,

            SIM_CLEAR   => SIM_CLEAR,
            SIM_CLOCK   => SIM_CLOCK
      );

      prot3: sim_protgen_synchpar
            generic map(
                  SIM_PIPELINE => 0,
                  Y1                      => Y1
            )
            port map(
                  IN_EN       => SUM2_VAL (sim_sigval_synchp...                    ⇒
...arVal),
                  -- when IN_EN high indicates that data for out...                 ⇒
...put is
                  -- valid at least until next rising edge of cl...                ⇒
...ock
                  IN_RDY      => SUM2_RDY,
                  Y1_VAL      => Y1_VAL,
                  Y1_RDY      => Y1_RDY,
                  SIM_CLOCK   => SIM_CLOCK,
                  SIM_CLEAR   => SIM_CLEAR
            );

end;
```

Apéndice C

Ejemplo de una sesión de trabajo con *Retiner*

C.1. Instalación y ejecución de *Retiner*

La versión actual de *Retiner* se distribuye mediante un archivo comprimido que contiene toda la estructura de directorios y ficheros necesarios para su correcto funcionamiento.

La instalación es tan sencilla como descomprimir dicho fichero en un cierto directorio que después añadiremos a la ruta por defecto de *Matlab* de forma recursiva (integrando automáticamente los subdirectorios), con objeto de hacer visible las funciones del entorno *Retiner* al ambiente por defecto de *Matlab*.

La ejecución de *Retiner* es tan sencilla como escribir `retiner` en la línea de órdenes de *Matlab*. Acto seguido aparecerá el interfaz de usuario de *Retiner*, listo para trabajar.

En entornos *Windows*, se encontrará también el archivo `retiner.bat`. La ejecución de este archivo se encarga de abrir una sesión *Matlab* y ejecutar automáticamente *Retiner*. Con todo, la manera recomendada de iniciar el entorno es la primera.

Para asegurar el perfecto funcionamiento de todas las partes y bibliotecas de *Retiner*, se realiza una comprobación exahustiva de todos los ficheros necesarios para su correcta ejecución, indicándose si se diera el caso, el elenco de los ficheros no encontrados.

C.2. Ejemplo de una sesión de trabajo con *Retiner*

A continuación se muestra una ruta a través de una sesión normal con *Retiner*, con objeto de resaltar su utilidad para diseñar y validar modelos de retinas, y como caso particular, modelos orientados al desarrollo de prótesis visuales.

neuroprótesis cortical para restaurar la ceguera. También se detallan o se hace más hincapié en algunos aspectos fundamentales no comentados en el apartado anterior.

Supuesto que hemos instalado convenientemente *Retiner* según el apartado C.1, y que disponemos de una webcam compatible con *Microsoft Video for Windows*, vamos a intentar diseñar una retina cuyos resultados puedan ser contrastados con datos biológicos reales, concretamente correspondientes a una retina de conejo.

C.2.1. Inicio de *Retiner* y selección de la entrada

Antes de iniciar *Retiner* y una vez nos encontremos en una sesión *Matlab*, conectaremos una *webcam* a nuestra computadora. Una vez realizada esta tarea, iniciamos *Retiner* según el apartado C.1. *Retiner* iniciará automáticamente nuestra *webcam*, procediento a mostrar una pequeña aplicación, de nombre *HSM–RI video driver* con una captura de la misma. Dicha aplicación contiene un

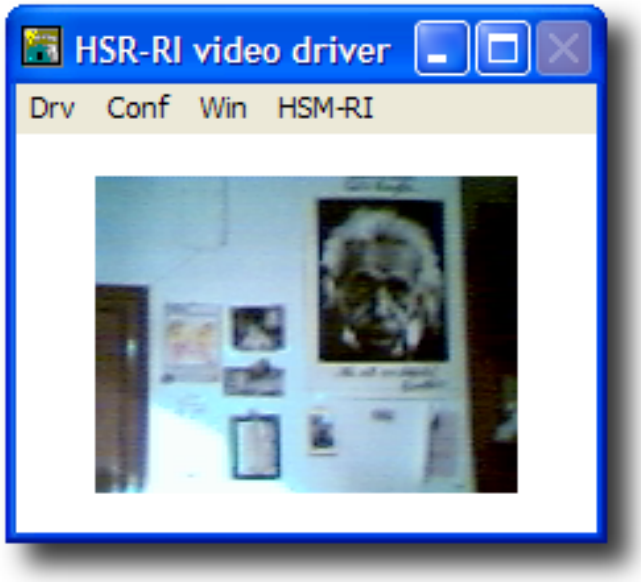

Figura C.1: Utilidad *HSM–RI* mostrando la captura proveniente de una sencilla *webcam*

menú con 4 entradas, las 3 primeras son heredadas directamente del sistema *Microsoft Video for Windows* y la tercera es de diseño propio. La lista de dichos submenús es la siguiente:

- `Drv`. Menú que muestra el número de drivers disponibles en su sistema *Windows* asociados con los dispositivos de video conectados a la computadora en ese momento. Puede haber más de 1 driver asociado a una misma videocámara.

- `Conf`. Menú para configurar los parámetros principales de la videocámara que soporta *VfW*, como:
  - Control de ganancia
  - Exposición
  - Mejoras de la imagen (Compensación de luz y color).
  - Control de frecuencia
  - Reflejar la imagen vertical u horizontalmente
  - Seleccinar nuevo driver de video
  - Resolución de la imagen en píxels (ancho $\times$ largo)
  - Profundidad y comprensión de píxels (RGB24, RGB555, ...)
  - Etc...
- `HSM-RI`. Accedemos desde este menú a ciertos parámetros del *API* de *VfW* que no encontramos en los submenús anteriores, como la desconexión del driver.

Nota El uso de una aplicación externa para realizar la conexión entre *Matlab–Retiner* y una videocámara utilizando la tecnología *Microsoft Video for Windows* y no otras como *Microsoft DirectX* está justificada por:

- La tecnología escogida está presente en todas las versiones de *Windows* y su *API* es estándar para todas ellas. El uso de la tecnología *DirecX*, concretamente *DirectShow*, aunque está generalmente más optimizada, no es una característica esencial para el funcionamiento de *Retiner*, ya que interesa más ver el resultado de la aplicación de los distintos filtros y funciones, que la propia captura de pantalla, y en esto, sendas tecnologías pueden ser consideradas igualmente eficientes. Además, el *API* de *DirectX* suele variar y añadir una dependencia pesada a las aplicaciones que dependen de él.
- Aunque la tecnología *Microsoft Video for Windows* es propia de los sistemas *Windows*, el sencillo *API* ha sido portado de forma satisfactoria a otros sistemas como *Linux*. También, estamos trabajando en un envoltorio para la aplicación *HSM–RI video driver*, que utilice el *API Video for Linux* de forma nativa en entornos *Linux*..
- Aunque *Mathworks* ha sacado al mercado un paquete de adquisición de imágenes, *Matlab Acquisition Toolbox* [46], que encapsula el comportamiento de *HSM–RI video driver*, éste paquete sólo está disponible a partir de la versión 7 de *Matlab* y solamente para plataformas *Windows*.
- La aplicación *HSM–RI video driver* es una modificación del software *Video for Matlab*, para adecuarlo a los propósitos de *Retiner* que se distribuye de forma gratuita mediante *Internet*.

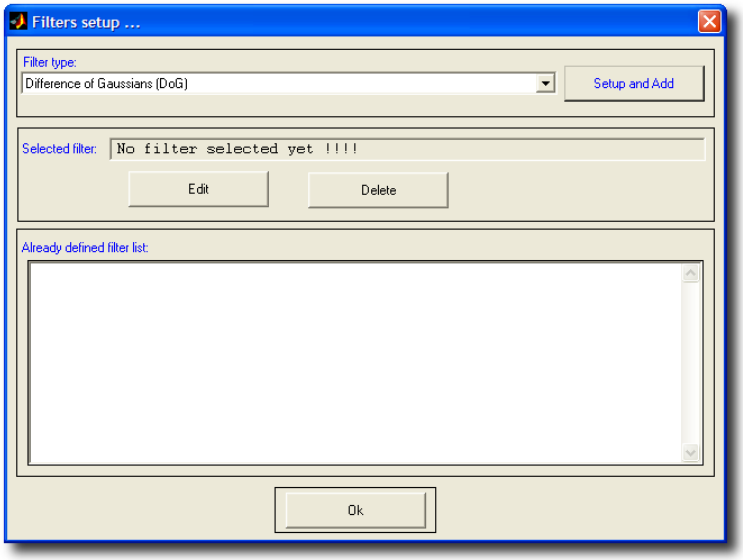

Figura C.2: Cuádro de diálogo `Filters setup...` sin ningún filtro.

C.2.2. Especificación del modelo

En el modelo de retina que vamos a simular, seleccionamos 1 para la ganancia de los fotorreceptores (ver figura 5.4) y seleccionamos el submenú `Filters setup...` del menú `Filters`. Esta utilidad está pensada para añadir y configurar los filtros estándar de *Retiner*, de forma asistida. Al comenzar, nos encontraremos que no existe nigún filtro definido, por lo que nos veremos algo similar a la figura C.2. Presionando sobre el botón `Setup and Add` podemos añadir distintos filtros para hacerlos accesibles al motor de simulación de *Retiner*. Los filtros *asistidos* que nos encontramos en la versión 1.0.0 de *Retiner* son los siguientes:

- *Difference of Gaussians (DoG)*. Que añade un filtro de tipo *Diferencia de Gausianas*.
- *Gaussian*. Con el que podemos añadir una gausiana.
- *LoG (Laplacian of Gaussian)*. Que añade un filtro de tipo *Laplaciana de Gausiana*.
- *Temporal Enhancement*. Que permite definir un filtro de realce temporal compatible con la biología y enfocado especialmente al diseño de retinas.
- *User defined filter*. Que, como reza su enunciado, nos brinda la opción de utilizar una expresión *Matlab* que será tratada como un filtro más dentro del entorno *Retiner*. La asistencia en la edición de estos filtros consiste

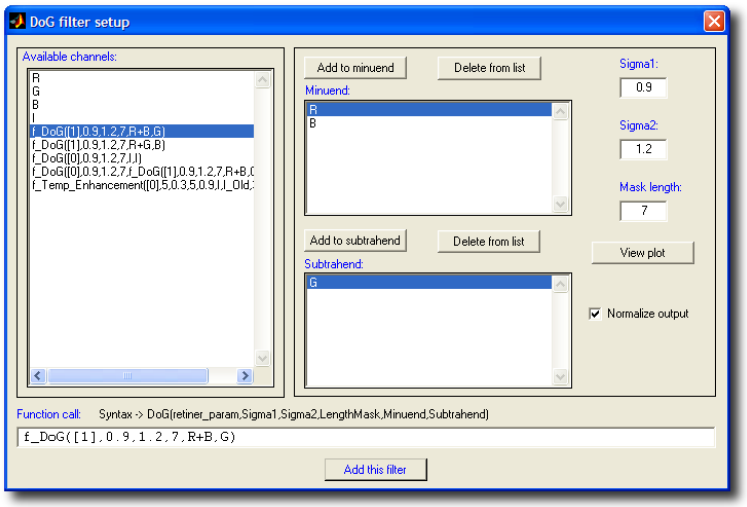

Figura C.3: Edición asistida de un filtro *Diferencia de Gausianas*.

sólo en un *parsing* o análisis de la expresión para asegurar que sea compatible con *Matlab* y con *Retiner*. En esta expresión se pueden usar todos los filtros ya definidos hasta ese momento y variables propias de *Retiner* (como los canáles cromáticos *R*, *G* y *B*) con tal de que, cuandren las dimensiones de todas las variables. Está permitido el uso de cualquier notación *Matlab* así como la llamada a otras funciones previamente definidas por el usuario.

Seleccionando el filtro *Diferencia de Gausianas* (*DoG*) y pulsando sobre `Setup and Add` para configurar y editar dicho filtro, accedemos a la utilidad para configurar esta clase de filtros, tal y como refleja la figura C.3.

El resto de los filtros asistidos disponibles desde *Retiner*, sigue el mismo esquema de configuración. No obstante, se detalla el filtro *DoG* por ser el que más opciones presenta. A la izquierda del cuadro de diálogo, la lista de nombre `Available channels`, contiene las variables predefinidas por *Retiner* relativas a los canáles cromáticos, *R*, *G*, *B* e *I*, para los canales rojo, verde, azul y el canal de intensidad; así como los filtros ya definidos por el usuario en formato función de *Matlab*. Todas estas variables pueden ustilizarse como entradas en el filtro que se esté editando. El asistente permite seleccionar las variables o filtros disponibles y añadirlos o borrarlos convenientemente al minuendo o sustraendo de la diferencia de gausianas mediante los pulsadores de la anterior figura C.3. Cada evento en el cuadro de diálogo se verá automáticamente reflejado en la parte inferior del asistente, donde se puede leer `Function call`. Allí se actualizan los parámetros de la función que estemos editando (*DoG* en este caso) y se muestra la sintaxis de dicha función, que en este caso responde a:

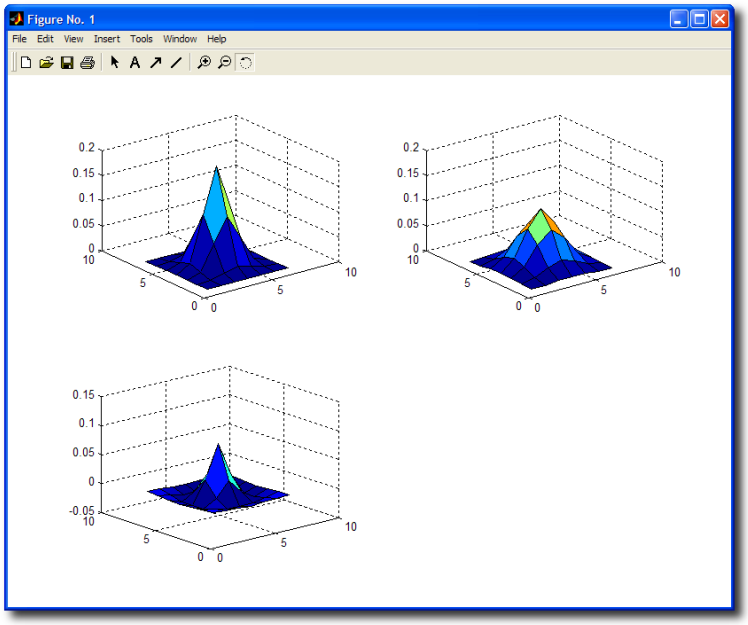

Figura C.4: Visualización de las gausianas y la diferencia de gausianas que ayuda a *sintonizar* mejor el filtro.

DoG`(retiner_param, Sigma1, Sigma2, LengthMask, Minuend, Subtrahend)`

y que después de configurar el nuevo filtro queda:

$$\mathbf{DoG}([1], 0.9, 1.2, 7, R + B, G) \tag{C.1}$$

El parámetro `retiner_param` está presente en todas las funciones nativas de *Retiner*. Se trata de una matriz con información extra sobre la actuación del filtro. En la versión 1.0.0 de *Retiner*, sólo puede tomar el valor $[0]$ o $[1]$ para indicar si se quiere o no, normalizar la salida del filtro en el intervalo $0 \ldots 255$, opción que se edita con el pulsador `Normalize output`.

También podemos editar, como es natural, las desviaciones típicas relativas a las gausianas que entran en juego en el filtro que nos ocupa. Generalmente, un filtrado *DoG* bioinspirado requiere que se conforme la forma *sombrero mexicano*. *Sintonizar* este filtro en una matriz de dimensiones reducidas (10×10 en este caso) puede ser complicado. Por eso se permite visualizar las dos gausianas que conforman la *DoG*, así como la propia *DoG*, tal y como se muestra en la figura C.4.

Con los asistentes de filtros sólo se puede definir una relación de suma entre las distintas variables de las entradas que queramos utilizar. En este caso, por ejemplo, tal y como se mostró en la ecuación C.1, se han utilizado las variables

R y B para el minuendo de la *DoG*. *Retiner* permite cambiar esta relación, por cualquier otra conforme con el lenguaje *Matlab* de forma semiasistida, ya que queda en manos del usuario la modificación directa de la llamada al filtro, pero no se actualizará su valor si no supera el *parsing* o análisis de la expresión por parte de *Retiner*. Podemos acceder a la edición manual del filtro selecionándolo en la lista de filtros disponibles del cuadro de diálogo de la figura C.2 y pulsando `Edit`.
El filtro de realce temporal bioinspirado, o `Temp_Enhancement`, cuya captura se muestra en la figura C.5 realiza las siguientes operaciones:

1. Efectúa la sustracción del sustraendo sobre el minuendo que se le indica, que puede ser como hemos visto, cualquier otra combinación de variables y filtros.
2. Tiene en cuenta los cambios positivos, negativos o de ambos signos para simular el comportamiento de las células ganglinares de tipo *ON*, *OFF* u *ON/OFF*. Parámetro que se edita en el cuadro `Transient Mode` de la figura.
3. Multiplica cada valor del resultado por el coeficiente de una matriz *fovea*, que puede ser definida en el própio asistente.
4. Realiza un filtrado gaussiano para suavizar los cambios y eliminar ruido.
5. Opcionalmente procede a normalizar la salida.

Este filtro, pretende modelar la *sensibilidad temporal variable* que presentan las retinas biológicas desde su centro hasta la periferia. *Retiner* lleva a cabo el realce de la actividad de la periferia mediante su realzado, manteniendo el nivel de actividad constante en la *fóvea*. Para ello, como se ha comentado, se multiplica cada coeficiente de la matriz de entrada por su elemento correspondiente de la matriz *fóvea* de *Retiner*, que se define como:

matriz fóvea Matriz que representa una función contínua formada de una gausiana centrada y truncada a 1 en el centro.

Los parámetros que pueden ser editados en el asistente para este filtro son los siguientes:

- `Extreme value`. Valor máximo de la matriz fóvea.
- `Fovea proportion`. Parámetro que configura el radio del centro foveal, cuyo valor es constante e igual a 1. Puede tomar valores entre 0 y 1, y representa el tanto por uno del radio foveal entre el centro de la matriz y la esquina más alejada de éste.
- `Gaussian blurring mask size`. Longitud de la máscara gausina empleada para suavizar el resultado. Esta matriz es cuadrada.
- `Gaussian blurring sigma`. Desviación típica de la anterior gausiana.

La fóvea puede visualizarse en 3 dimensiones con objeto de ajustarla mejor a los requisitos del diseñador. Para ello, accionando el pulsador `View foveated shape`, de la C.5, obtenemos la matriz fóvea que muestra la figura C.6. También es posible, como en el resto de los filtros, visualizar las máscaras de convolución que toman parte en el filtro.

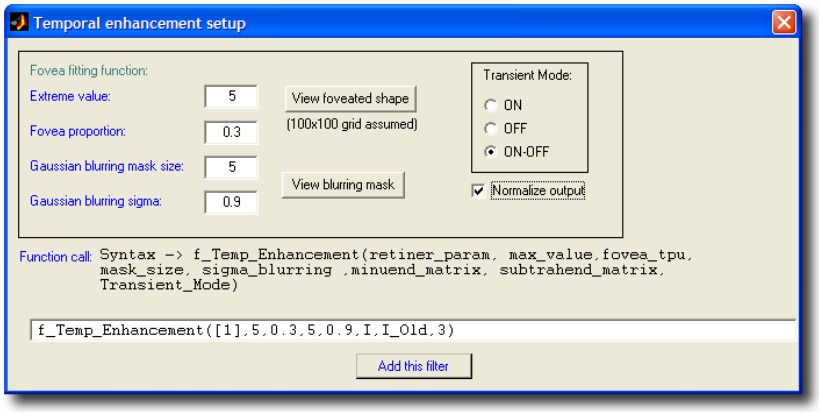

Figura C.5: Diseño asistido del filtro de realce temporal bioinspirado.

Una vez definido el elenco de filtros, *Retiner* los nombra comenzando por $F1$ hasta FN, siendo N el número de filtros definido, tal y como muestra la figura C.7. Se puede observar que se han definido 5 filtros tipo *DoG*, de los cuales 4 se han utilizado para definir la combinación. Los filtros definidos y la combinación a utilizar son siguientes:

$$R \tag{C.2}$$
$$G \tag{C.3}$$
$$B \tag{C.4}$$
$$I \tag{C.5}$$
$$F1 = DoG([1], 0.9, 1.2, 7, R+B, G) \tag{C.6}$$
$$F2 = DoG([1], 0.9, 1.2, 7, R+G, B) \tag{C.7}$$
$$F3 = DoG([0], 0.9, 1.2, 7, I, I) \tag{C.8}$$
$$F4 = DoG([0], 0.9, 1.2, 7, F1+G, R) \tag{C.9}$$
$$F5 = Temp\_Enhancement([0], 5, 0.3, 5, 0.9, I, I\_Old, 3) \tag{C.10}$$

$$\textbf{Modelo} = (2F1 + F2 + F3 + F5)/5 \tag{C.11}$$

C.2.3. Configuración de los campos receptivos

El proyecto *Cortivis* [47], en el que se ha fundamentado este trabajo de tesis doctoral, comenzó empleando datos biológicos tomados a partir de experimen-

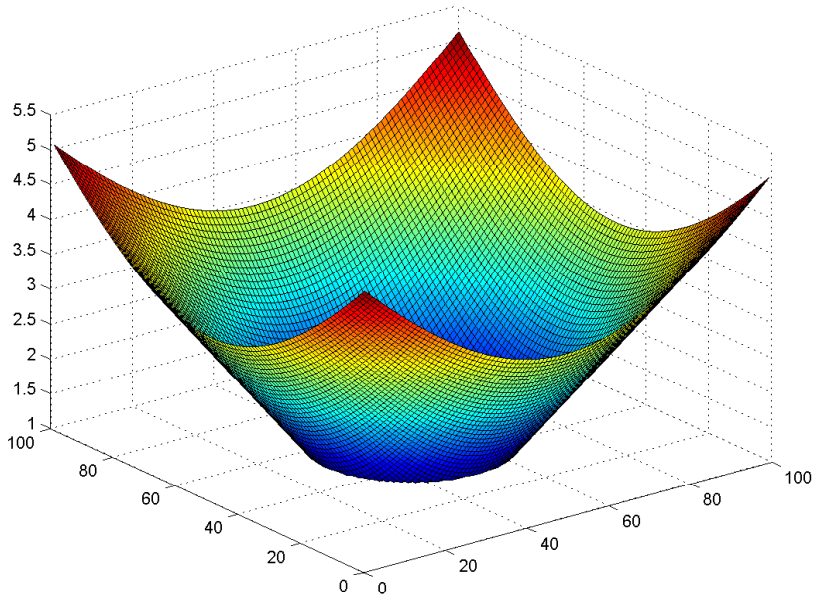

Figura C.6: Representación de la función que configura la actuación de la fovea, dando un peso mayor a la componente temporal en la zona extrafoveal.

tos que utilizaban la matriz de 10×10 microelectrodos de la Universidad *Utah* [48, 17]. Se tienen registros de numerosos experimentos relativos a la excitación neuronal de las células de distintas retinas animales (conejos, gatos, etc...) y también unos pocos registros en retinas humanas. Es por este motivo que las dimensiones por defecto de la matriz de microelectrodos sean de 10×10, que serán también las utilizadas para este ejemplo.
Por ello, seleccionamos dichas dimensiones en el menú `Receptive fields`, submenú `Activity Matrix size...` de la figura 5.6 anteriormente referida.
Elegimos campos receptivos rectangulares no solapantes, opción ésta por defecto como se muestra en la figura 5.6. No obstante, *Retiner* ofrece la posibilidad de definir otros campos receptivos más sintonizados con la biología, mediante la utilidad externa `Receptive field definition` activada mediante el submenú `Receptive fields⇝Custom...⇝Define...` que permite definir:

- Campos receptivos rectangulares. Donde es posible modificar su altura, anchura y orientación.
- Campos receptivos cuadrados. Análoga a la anterior pero usando cuadrados.
- Campos receptivos elípticos. Con anchura y altura modificables.

- Campos receptivos circulares. Donde podemos seleccionar un conjunto de círculos cuyos radios pueden incrementar de centro a periferia, con una relacción que puede ser lineal cuadrática.

La figura C.8 muestra un ejemplo de edición de un campo receptivo circular de radio linealmente variable entre 8 unidades en su centro (fóvea) hasta 19 unidades en su periferia. Este es el modelo campo receptivo potencialmente más parecido a la biología ya que da cuenta de la distribución no constante de fotorreceptores a lo largo de la retina.
Los datos generados correspondientes al campo receptivo seleccionado, son guardados en formato *ASCII* con lo que pueden ser modificados con cualquier editor sencillo de texto plano. El formato de dicho archivo es el siguiente:

```
<Número_de_filas>×<Número_de_Columnas>
<Columna_del_electrodoN,Fila_del_electrodoN>=
···[[pixel_x,pixel_y]],...
```

Es decir, para la matriz de microelectrodos que nos ocupa y supuesto que queremos campos cuadrados de lado 3 solapantes al $66\,\%$ se tendría algo simular a lo siguiente:

10×10
...
$3,4 = [2,3][2,4][2,5][3,3][3,4],[3,5][4,3][4,4][4,5]$
$3,5 = [2,4][2,5][2,6][3,4][3,5],[3,6][4,4][4,5][4,6]$
...

Dentro de este menú, tal y como se mostró en el apartado 5.2.5, es posible guardar el resultado de la matriz de actividad en formato *AVI*. Dicha salida puede ser suavizada mediante cualquiera de las funciones del submenú `Receptive fields⇝Graded output method...`, siendo como comentamos, el emborronamiento gausiano, el método más próximo a la biología.

C.2.4. Configuración Modelo neuronal *Retiner*

Retiner utiliza un modelo neuronal de tipo *integra y dispara* para obtener la secuencia de trenes de impulsos ligadas a cada microelectrodo. Se accede a la edición de dichos parámetros mediante el submenú `Electrode stimulation ⇝Firing configuration...` que abre el cuadro de diálogo mostrado en la figura C.9.

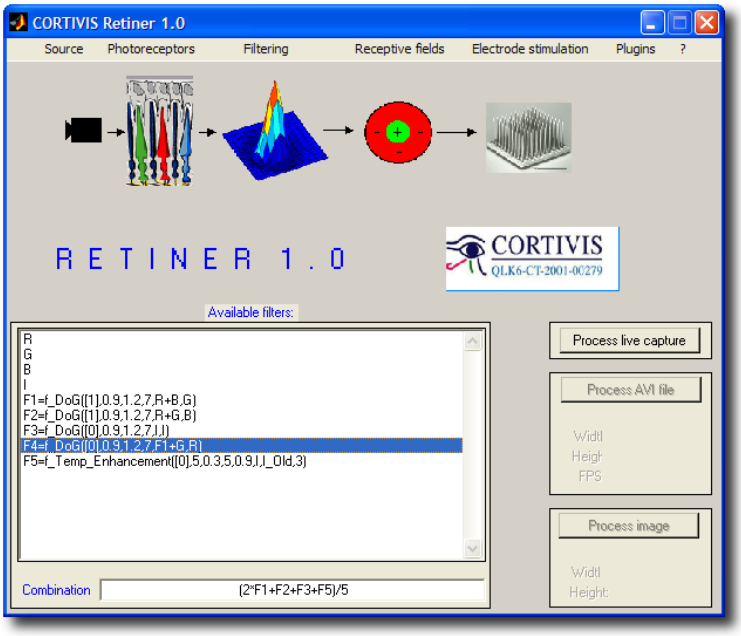

Figura C.7: Interfaz principal de *Retiner* una vez definidos todos los filtros y la combinación de ellos que se utilizará

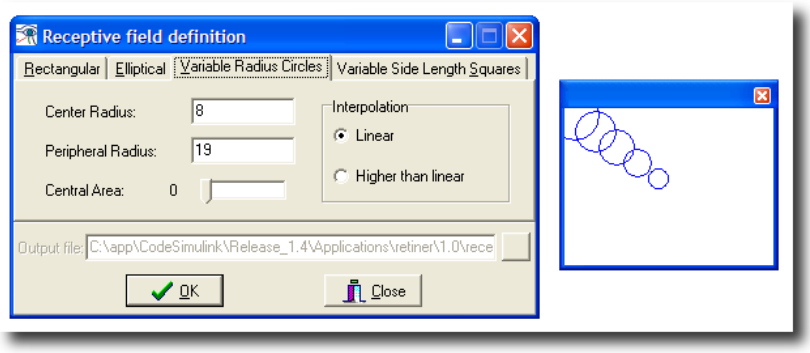

Figura C.8: Definición de los campos receptivos de la retina mediante la utilidad `Receptive field definition` de *Retiner*

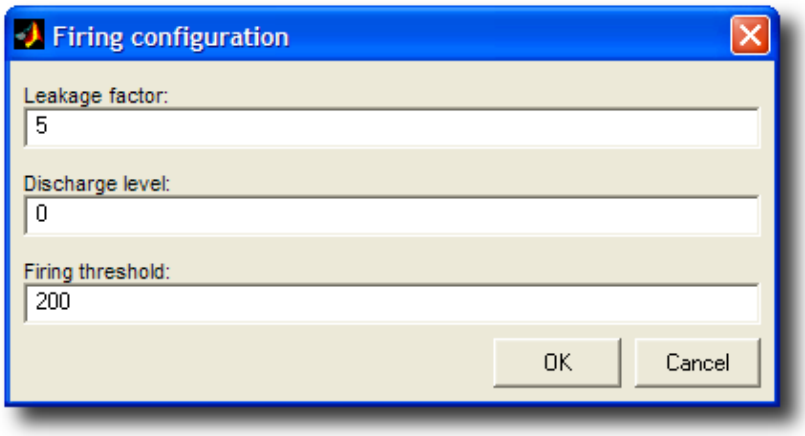

Figura C.9: Edición de los parámetros del modelo neuronal tipo *integra y dispara*

Apéndice D

Relacción de Acrónimos

API Application Programming Interface

CIL Common Intermediate Language

FPGA Field Programmable Gate Array

HDL Hardware Description Language

PCI Peripheral Component Interconnect

VHDL VHSIC-HDL, Very High Speed Integrated Circuit-HDL

Índice alfabético

Bibliografía

[1] E.Ros, F.J.Pelayo, A.Prieto, and B. del Pino. Ingeniería neuromórfica: el papel del hardware reconfigurable. In *II Jornadas sobre Computación Reconfigurable y Aplicaciones (JCRA'02)*, pages 89–92, Almuñécar (Granada), 18–20 de Septiembre 2002. ISBN: 84-699-9448-4. 2, 48

[2] G. De Micheli. Computer-aided hardware-software codesign. *IEEE Micro*, 25(6):10–16, Agosto 2005. ISBN: 0272-1732. 13

[3] Sandra Sinclair. *How animals see : other visions of our world*. Facts on File Publications, 1985. ISBN: 0-87196-273-X. 30

[4] John Downer. *Supersense. Perception in the Animal World*. Holt and Co., New York, 1988. ISBN: 0805010874. 30

[5] John E. Dowling. *The retina: an approachable part of the brain.* Belknap Press, Septiembre 1987. ISBN: 0-674-76680-6. 32

[6] Elga Kolb, Eduardo Fernández, and Ralph Nelson. Página principal. `http://webvision.med.utah.edu`. 32

[7] Bach M. The freiburg visual acuity test–automatic measurement of visual acuity. *Optometry and vision science*, 73:49–53, 1 January 1996. ISSN: 1040-5488. 34

[8] S. Ramón y Cajal. *La rétine des vertebrés*, volume IX-1. La Cellule, 1892. 35

[9] Kolb H., Fernandez E., Schouten J., Ahnelt P., Linberg KA., and Fisher SK. Are there three types of horizontal cell in the human retina? *Journal of Comparative Neurology*, 343:370–386, 1994. ISSN: 0021-9967. 40

[10] Ahnelt P. and Kolb H. Horizontal cells and cone photoreceptors in human retina: a golgi-electron microscopic study of spectral connectivity. *Journal of Comparative Neurology*, 343:370–386, 1994. ISSN: 0021-9967. 40

[11] Daniel V. Palanker, Philip Huie, Alexander B. Vankov, Yev Freyvert, Harvey Fishman, Michael F. Marmor, and Mark S. Blumenkranz. Attracting retinal cells to electrodes for high-resolution stimulation. *Lasers in Surgery: Advanced Characterization, Therapeutics, and Systems XIV.*, 5314:306–314, Julio 2004. 44

[12] Mark S. Humayun, Eugene de Juan Jr., and James D. Weiland. Pattern electrical stimulation of the human retina. *Vision Research*, 39:2569–2576, 1999. ISSN: 0042-6989. 44

[13] Veraart C., Wanet-Defalque M.C., Gérard B., Vanlierde A., and Delbeke J. Pattern recognition with the optic nerve visual prosthesis. *Artificial Orgnas*, 27(11):996–1004, 2003. ISSN 1434-7229. 45, 46

[14] Dobelle W.H., Mladejovsky M.G., and Girvin J.P. Artificial vision for the blind: electrical stimulation of visual cortex offers hope for a functional prothesis. *Science*, 183:440–444, 1974. 47

[15] Dobelle WH, Quest DO, Antunes JL, Roberts TS, and Girvin JP. Artificial vision for the blind by electrical stimulation of the visual cortex. *Neurosurgery*, 5(4):521–527, 1979. 47

[16] Dobelle W.H. Artificial vision for the blind by connecting a television camera to the visual cortex. *American Society of Artificial Internal Organs (ASAIO)*, 46:3–9, 2000. ISSN: 1058-2916. 47

[17] E. M. Maynard, C. T. Nordhausen, and R. A. Normann. The Utah intracortical electrode array: a recording structure for potential brain-computer interfaces. *Clinical Neurophysiology*, 102:228–239, 1997. 48, 58, 78, 80, 177

[18] Misha Mahowald. *An Analog VLSI System for Stereoscopic Vision*. Kluwer Academic Publishers, 1994. 49

[19] Brian A. Wandell. *Foundations of Vision*. Stanford University, 1995. ISBN: 0-87893-853-2. 49

[20] Jordan T. Berry M., Brivanlou I. Anticipation of moving stimuli by the retina. *Nature (Lond)*, 398:334338, 1999. 49

[21] M. J. Schnitzer and M. Meister. Multineuronal firing patterns in the signal from eye to brain. *Neuron*, 37:499–511, 2003. ISSN: 0896-6273. 49

[22] Stephen A. Baccus y Markus Meister Bence P. Ölveczky. Segregation of object and background motion in the retina. *Nature*, 423:401–408, 2003. 49

[23] M. y M. J. Berry Meister. The neural code of the retina. *Neuron*, 22:435–450, 1999. 49

[24] Laurent Itti and Christof Koch. Computational modelling of visualsual attention. *Nature Reviews Neuroscience*, 2:194–203, March 2001. ISSN: 1097-6256. 50

[25] F.J.Pelayo, S. Romero, C.Morillas, A.Martinez, E.Ros, and E.Fernandez. Translating image sequences into spikes patterns for cortical neurostimulation. *Neurocomputing*, 58-60:885–892, 2004. ISSN: 0925-2312. 55

[26] Christian A. Morillas, Samuel F. Romero, Antonio Martínez, Francisco J. Pelayo, and Eduardo Fernández. A computational tool to test neuromorphic encoding schemes for visual neuroprostheses. *Lecture Notes in Computer Science*, 3512:510–517, 2005. ISSN: 0302-9743, ISBN: 3-540-26208-3. 57, 62

[27] Samuel Romero, Francisco J. Pelayo, Christian A. Morillas, Antonio Martínez, and Eduardo Fernández. *Reconfigurable Retina-like Preprocessing Platform for Cortical Visual Neuroprostheses*, volume 3, chapter Neural Engineering: Neuro-Nanotechnology - Biorobotics, Artificial Implants and Neural Prosthesis. IEEE Press Series on Biomedical Engineering, metin akay edition, 2005. ISBN: 0-471-68023-0. 62

[28] Christian A. Morillas, Samuel F. Romero, Antonio Martínez, Francisco J. Pelayo, Eduardo Ros, and Eduardo Fernández. A design framework to model retinas. *Biosystems, Computational Neuroscience: Trends in Research 2004*, 2004. ISSN: 0303-2647. 62

[29] Antonio Martínez, Leonardo M. Reyneri, Francisco J. Pelayo, Samuel F. Romero, Christian A. Morillas, and Begoña Pino. Automatic generation of bio-inspired retina-like processing hardware. *Lecture Notes in Computer Science*, 3512:527–533, 2005. ISSN: 0302-9743, ISBN: 3-540-26208-3. 63

[30] A. Martínez, F.J. Pelayo, C. Morillas, S. Romero, R. Carrillo, and B. Pino. Generador automático de sistemas bioinspirados de visión en hardware reconfigurable. In *IV Jornadas de Computación Reconfigurable y Aplicaciones (JCRA'04), Barcelona*, 13-15 de Septiembre 2004. ISSN: 84-688-7667-4. 63

[31] Antonio Martínez, Francisco J. Pelayo, Christian A. Morillas, Leonardo M. Reyneri, and Samuel Romero. Automatic synthesis of vision processors on reconfigurable hardware. In *V Jornadas de Computación Reconfigurable y Aplicaciones (JCRA'05) dentro del I Congreso Español De Informática (CEDI'2005)*, pages 179–184, 13-16 de Septiembre 2005. ISBN: 84-9732-439-0. 63

[32] Antonio Martínez Álvarez, Leonardo Maria Reyneri, and Francisco J. Pelayo Valle. Automatic synthesis of data–flow systems using a high level codesign tool. application to vision processors. In *aceptado para el congreso MELECON'2006, The 13th IEEE Mediterranean Electrotechnical Conference*, 16-19 de Mayo 2006. 63

[33] Diane Barlow Close, Arnold D. Robbins, Paul H. Rubin, Richard Stallm, and Piet van Oostrum. *The AWK Manual*, 1.0 edition, December 1995. 65

[34] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, third edition edition, July 2000. ISBN: 0-596-00027-8. 65

[35] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification*. Addison-Wesley, third edition edition, May 2005. ISBN 0-321-24678-0. 65

[36] Jesse Liberty. *Programming C#*. O'REILLY, July 2003. ISBN: 0-569-00489-3. 65

[37] Niel M. Bornstein Edd Dumbill. *Mono: A Developer's Notebook*. O'REILLY, July 2004. ISBN: 0-596-00792-2. 67

[38] Página web principal del proyecto mono. http://www.mono-project.com. 68

[39] Página web principal de *Matlab*. http://www.mathworks.com/products/matlab. 72

[40] Pal Rujan y Josef Ammermüller Martin Grescher, Markus Bongard. Retinal ganglion cell synchronization by fixational eye movements improves feature estimation. *Nature Neuroscience*, 5(4):341–347, 4 2002. ISSN: 1097-6256. 81

[41] Página web principal del producto de Celoxica PixelStrems. http://www.celoxica.com/products/pxs/default.asp. 87

[42] Kenneth R. Castleman. *Digital Image Processing*. Prentice Hall, Inc., 1996. ISBN: 0-13-211467-4. 92

[43] D. Ridgeway. *The Programmable Logic Data Book*, chapter Designing Complex 2-Dimensional. Convolution Filters. Xilinx, 1994. 94

[44] IEEE Computer Society. *IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std 754-1985*, 1985. 114

[45] F.J. Pelayo, C. Morillas, A. Martínez, S. Romero, and B. Pino E. Ros. A reconfigurable machine to model the first stages of the human visual pathway. In *Actas de las III jornadas sobre computación reconfigurable y aplicaciones (jcra'2003)*, pages 173–182, Madrid, 10-12 de Septiembre 2003. ISBN: 84-60099-28-8. 132

[46] Página web principal de *Image Acquisition Toolbox* para *Matlab*. http://www.mathworks.com/products/imaq. 171

[47] Página web principal del proyecto de investigación europeo CORTIVIS. (QLK6 CT 2001 00279). http://cortivis.umh.es. 176

[48] R. A. Normann, E. M. Maynard, K. S. Guillory, and D. J. Warren. Cortical implants for the blind. *IEEE Spectrum*, 33:54–59, May 1996. 177