VNIVERSITAT DE VALÈNCIA

UNIVERSITAT DE VALÈNCIA

Sistema de visión basado en procesado guiado por cambios y lógica reconfigurable para el análisis de movimiento de alta velocidad

Memoria para optar al grado de doctor presentada
al Departamento de Informática
Escuela Técnica Superior de Ingeniería

Julio César Sosa Savedra

Dirección

Dr. Fernando Pardo Carpio
Dr. Jose Antonio Boluda Grau

A la memoria de mi padre.
A mi familia.

Índice

Índice de cuadros

Índice de figuras

Agradecimientos

Este trabajo fue un gran reto para mi superación personal y profesional. No habría podido lograrlo sin la ayuda de muchas personas, tanto familiares como investigadores del área en la cual desarrolle mi trabajo. A esas personas les dedico las siguientes palabras.

A mi familia que siempre fue un aliciente para lograr concluir esta meta. Mis padres quienes me dieron la vida y la formación como persona. Mi esposa que siempre estuvo a mi lado apoyándome y mis hijos a quienes les robe valioso tiempo para dedicárselo a mi trabajo de tesis.

A mis directores de tesis, Dr. José A. Boluda Grau y el Dr. Fernando Pardo Carpio, que además de brindarme su apoyo técnico también me ofrecieron su amistad. Ellos me recibieron calurosamente en su equipo de trabajo, me animaron en todo momento e influyeron en mí para dar lo mejor.

A todos ellos les agradezco infinitamente todo su apoyo para concluir esta meta que sin duda fue una etapa trascendental en mi vida.

Es sano agradecer a todos aquellas instituciones que contribuyeron, de una u otra forma, financiando el presente trabajo de investigación. En primer lugar está el Instituto Politécnico Nacional, por la beca otorgada para mis estudios de doctorado. También al MCyT por su financiación del proyecto TIC 2001-3546, dirigido por el Dr. Fernando Pardo Carpio, y a la Universidad de Valencia por los proyectos de acciones especiales, UV-AE-20050206 y 20060242, autorizados al Dr. Fernando Pardo Carpio.

Resumen

El movimiento es la principal fuente de información que nos ayuda a determinar la forma y estructura de los objetos percibidos en nuestro entorno. De esta manera, muchas técnicas para la estimación del campo de velocidad son componentes indispensables para aplicaciones de visión. El campo de velocidad puede ser utilizado para una gran cantidad de aplicaciones, tales como: reconstrucción 3D, reconstrucción de imágenes afectadas por ruido, video compresión, segmentación por movimiento, seguimiento, navegación autónoma y estimación del tiempo al impacto.

Sin embargo, la estimación del movimiento es una aplicación que demanda una gran cantidad de recursos y este hecho viene a ser un cuello de botella cuando se requiere un procesamiento en tiempo real.

El cálculo del flujo óptico consiste en la estimación del campo del movimiento aparente 2D dentro de una secuencia de imágenes. De esta manera, cada píxel tiene asociado comúnmente un vector velocidad. En la literatura, existe una gran cantidad de estrategias para calcular el flujo óptico. De todos los métodos el basado en el gradiente y el basado en la correlación son los más ampliamente utilizados. Sin embargo, la principal restricción para la implementación en tiempo real de dichos algoritmos es sin duda la gran cantidad de información a procesar y el elevado costo computacional que representan el cálculo de dichos algoritmos. Ambas restricciones han sido abordadas muchas veces, desde un punto de vista de una reducción selectiva de la información, combinando sensores especiales con arquitecturas embebidas desarrolladas sobre FPGA's. Algunos otros trabajos utilizan arquitecturas en pipeline implementadas sobre FPGA's, computadoras paralelas y otros trabajos utilizando estaciones de trabajo.

El método clásico utilizado para el análisis de una secuencia de imágenes normalmente procesa todos los píxeles de la imagen y de todas las imágenes. Para el cálculo del flujo óptico las derivadas espaciales y temporales son calculadas para todas los píxeles de la imagen, a pesar de que las imágenes pudieron haber sufrido sólo pequeños cambios entre pares de imágenes consecutivas. En este trabajo se propone una estrategia de procesado guiado por los cambios existentes entre dos imágenes.

La finalidad de la estrategia es reducir el tiempo de procesado considerando que normalmente sólo existen pequeños cambios entre imágenes consecutivas, especialmente si la frecuencia de muestreo es alta, logrando así una reducción de tiempo. Por esta razón es necesario utilizar una arquitectura del tipo flujo de datos.

En una arquitectura de flujo de datos se ejecuta uno o varios comandos dependiendo de la información disponible en el proceso. Adicionalmente, una instrucción estará lista para su ejecución cuando la información necesaria esté disponible. Cada instrucción, dentro de la arquitectura flujo de datos, contiene los valores de la variable y la ejecución de la instrucción no afecta en ningún momento a otra instrucción. La arquitectura del tipo flujo de datos tiene algunas particularidades bien definidas. Por ejemplo no requieren tener memoria compartida, no requiere un contador de programa, ni un sistema de control principal. Por otro lado, esta arquitectura requiere un circuito que detecte datos disponibles, un circuito que verifique los datos e instrucciones y un circuito que ejecute, si es posible, las instrucciones asíncronamente.

Otra consideración hecha en este trabajo es utilizar una tarjeta de desarrollo PCI para implementar el diseño. El bus PCI es utilizado como un estándar de interconexión entre los diferentes módulos que integran el sistema de visión, tales como: la cámara, una computadora principal y la tarjeta de desarrollo. Es necesario utilizar una computadora principal debido a que los resultados obtenidos por este trabajo serán utilizados en futuras aplicaciones, especialmente en navegación autónoma.

La estrategia de procesado guiado por cambios, presentada en este trabajo, es implementada en una nueva arquitectura que permite una reducción en el tiempo de cálculo del flujo óptico. La desventaja que presenta esta arquitectura es que existe una pequeña pérdida en la precisión y una menor densidad en el campo del flujo óptico. Sin embargo, la reducción del tiempo de procesado compensa las desventajas.

Abstract

Motion is a prime source of information for determining the shape and structure of objects perceived in our environment. Therefore, many techniques for estimating the velocity field are indispensable component of vision applications. The velocity field may then be used for applications such as three-dimensional reconstruction, reconstruction of images affected by noise, video compression, segmentation from motion, tracking, robot navigation and time-to-collision estimation.

However, motion estimation is a high processing power demanding application that may become a critical bottleneck when real-time constrain are required.

The optical-flow computation consist on the estimation of the apparent 2D movement field in the image sequence. In this way, each pixel has an associated velocity vector commonly. There are many strategies for optical flow computation in the literature. Among these methods, the gradient-based and the correlation-based approaches are the two most widely used techniques. But, the main constrains for real-time implementation of these algorithms are the large amount of data to be processed and the high computational cost of the algorithms involved. Both restrictions have been approached many times from the point of view of the selective reduction of the information combining specialized sensors with custom FPGA embedded architectures. Some others works have used pipeline architectures on FPGA, parallel computer and work station.

In this work is presented an improved architecture that uses several tools and strategies of design. The classical approach for image sequence analysis usually involves full image processing. In the optical flow computation the spatial and temporal derivatives are calculated for all pixels and all images, despite the fact that images could have suffered minor changes from one frame to the next. Here is proposed the change-driven image processing strategy. The strategy is to reduced the processing time realizing that images usually change little from frame to frame, especially if the acquisition time is short.The reduction is accomplished executing the instructions only for the pixels which have changed between two consecutive images. For this, it is used a data flow architecture.

The data flow architecture executes a command or several commands depending of the available information in the process. Additionally an instruction will be ready for its execution when the necessary information is available. Every instruction, inside of the data flow architecture, has the values of the variables and the instruction execution does not affect another instruction. The data flow architecture has some particularities. The architecture does not need to have shared memory, a program counter and a principal control system. For another hand, it is needed a circuit to detect the available data, a circuit that verifies data and instructions and finally if is possible a circuit that executes instructions asynchronously.

Another consideration is using a PCI development board to implement the design. The PCI bus is an interconnection standard between the modules of vision system such as: the camera, the principal computer and the development board. It is necessary to have a principal computer, because the results of this work will be use in future applications, especially in robot navigation.

The change-driven image processing strategy presented at this work allows the implementation of a new architecture for speeding-up the optical-flow computation. The disadvantage is that it has a small lost in precision and density of the optical flow. But in certain way, the speed-up to processing compensates this small lost.

Prólogo

El origen del presente trabajo de investigación parte de la confluencia de varias líneas de investigación en el seno del grupo de Tecnologías y Arquitecturas de la PErcepción por Computador (TAPEC), del Departamento de Informática de la Escuela Técnica Superior de Ingeniería de la Universidad de Valencia. Desde el año de 1995 ya existía cierta experiencia en el uso de dispositivos lógicos programables y en el área de percepción por computador (visión artificial). Dichas nociones se utilizaron en diversos proyectos de investigación que posteriormente convergieron en las tesis doctorales del Dr. Fernando Pardo Carpio y del Dr. José Antonio Boluda Grau, ambos directores del presente trabajo de investigación. La última tesis doctoral, titulada *Arquitectura de procesamiento de imágenes basada en lógica reconfigurable para navegación de vehículos autónomos con visión foveal*, representa la línea de investigación que más coincide con el trabajo aquí desarrollado.

El presente trabajo de investigación se concibe en el proyecto del MCyT titulado *Análisis de movimiento de alta velocidad mediante el uso de imágenes foveales y visión binocular entrelazada* (TIC2001-3546). En el trabajo se utilizan las imágenes foveales con la finalidad de aprovechar una de sus características, en particular el beneficio de la disminución de datos a procesar. La visión foveal tiene una representación en la que la resolución es mayor en el centro de la imagen y disminuye conforme se aleja del centro de la imagen, permitiendo una reducción considerable de la información a procesar. Por lo tanto se reduce la información a procesar de manera selectiva, procesándose en mayor cantidad los píxeles que se encuentran localizados en el centro de la imagen y muy pocos píxeles localizados en la periferia. Este hecho indica que existe una pérdida de información de la zona localizada en la periferia de la imagen, a pesar de que existiera algún tipo de movimiento, algo que es permitido en algunos sistemas de visión pero en otros sistemas este hecho puede no ser aceptable.

En otro trabajo se realiza un análisis de movimiento de alta velocidad, el nombre del proyecto era *Técnicas empotradas para el procesado en tiempo real de la información perceptual de robots móviles* (UV-AE-20050206), proyecto de acciones

especiales de la Universidad de Valencia. La intención fundamental del trabajo fue realizar un procesamiento en tiempo real para un robot autónomo. Este proyecto permitió profundizar más en técnicas de procesado utilizando arquitecturas especiales para un procesado en tiempo real.

Durante el desarrollo de los trabajos de investigación surgen varias ideas y propuestas, todo en función de percibir como trabaja el sistema de visión biológico. Se entiende que el envío de la información visual por parte del ojo humano se realiza de forma asíncrona, por parte de cada píxel de forma individual; sin embargo, prácticamente toda la visión artificial actual está basada en la adquisición síncrona de imágenes completas. La principal ventaja del modelo biológico es la capacidad de poder reaccionar ante estímulos en el instante mismo en que se producen y no a intervalos fijos preestablecidos como en el modelo clásico de tratamiento de imágenes.

Finalmente, continuando con esta idea del procesado de la información asíncrona, se realiza un estudio más detallado en el proyecto titulado *Desarrollo de técnicas de análisis de imágenes basadas en procesado por cambios y su aplicación al reciclado automático* (UV-AE-20060242). Con este trabajo se obtienen resultados concluyentes en los que se aprecia una ventaja de la técnica propuesta, denominada procesado guiado por cambios, sobre técnicas comúnmente utilizadas en las que se procesan todos los píxeles de la imagen. Durante este proyecto se ha realizado el diseño de los algoritmos en un dispositivo reconfigurable, su depuración y su experimentación. Así mismo, se realizó la integración del módulo reconfigurable en una tarjeta de desarrollo que trabaja conectada al bus PCI. Esto supone estudio y el diseño de componentes adicionales para poder realizar la comunicación con la interfaz PCI.

De manera adicional, a partir de los resultados del presente trabajo de investigación, se ha planteado la extensión del trabajo realizado en el proyecto de reciente aprobación titulado *Desarrollo de técnicas y sensor para visión asíncrona guiada por cambios para el análisis de movimiento a muy alta velocidad* (TEC2006-08130/MIC).

El trabajo y estudio realizado en los diversos campos implicados en el presente trabajo de investigación han sido muy enriquecedores para el autor. La intención final es que la investigación realizada en este trabajo responda a los objetivos planteados desde un inicio.

Parte I

Introducción y fundamentos

Capítulo 1

Introducción y contenido

1.1. Introducción

La idea de desarrollar sistemas inteligentes, dotados con funciones y capacidades similares a las de los humanos, puede ser remontado a la antigüedad. En pos de ese objetivo, surge la necesidad de dotar a los sistemas inteligentes capacidades sensoriales parecidas a los del ser humano. Sin lugar a duda la percepción visual es el sistema más importante del ser humano pues permite el conocimiento espacial de un entorno relativamente amplio y a una resolución considerablemente alta. El sistema de visión percibe el movimiento con muy poco esfuerzo y, durante la percepción del movimiento, procesa escenas dinámicas de forma natural, identifica sin dificultad los objetos que componen una escena y detecta el comportamiento temporal de los objetos que participan en la escena.

La capacidad de la percepción visual es muy amplia: ayuda en la estimación del movimiento relativo entre objetos, estimación del movimiento propio, procesos de orientación, discriminación entre distintos objetos e incluso en procesos de reconocimiento ya sea para seguir algún objeto dentro de una escena o para ubicar objetos particulares en la misma. En resumen, en cualquier sistema de visión, es indispensable efectuar un análisis del movimiento de los objetos dentro de la escena.

De este modo, desde hace muchos años se ha intentado imitar a los sistemas de visión biológicos, debido a su perfección, sin embargo, ha resultado ser más complicado de lo que parecía en un principio. Debido al poco éxito logrado se cedió terreno a sistemas activos de percepción de distancias como los de ultrasonido o el láser. Sin embargo, paralelamente, se han realizado estudios, con mayor interés en los últimos

años, en el área de visión artificial pero con distintas visiones y objetivos. Así pues, nacen distintos enfoques que abordan el estudio de los sistemas, los cuales son:

- **Empírico.** Busca determinar los mecanismos por los cuales los humanos y otros animales son capaces de ver y de esta manera reproducirlos e implantarlos en sistemas artificiales.
- **Normativo.** Intenta determinar cual debe ser el mecanismo de percepción visual óptimo para realizar alguna tarea específica aunque no tenga alguna relación con mecanismos biológicos.
- **Teórico.** Pretende identificar cuales pueden ser los posibles mecanismos que permitan desarrollar sistemas de percepción visual.

El enfoque empírico ha llevado a numerosos estudios en los campos de la biología, psicología y fisiología, hallando modelos fundamentalmente analíticos que establecen las relaciones entre las distintas partes del cerebro y su participación en los procesos visuales. Si bien no proporcionan soluciones explícitas al problema de percepción artificial, sirven de referencia y posible justificación de determinadas propuestas.

No obstante, los esfuerzos hechos en el campo de la visión artificial quedan de manera general englobados en los enfoques normativo y teórico. En estos, no resulta imprescindible que la alternativa hallada se corresponda con la biológica, sino que proporcione un planteamiento correcto y eficaz en cuanto al cómputo se refiere. Esta actitud es llevada al extremo en el enfoque normativo donde no se plantea la generalidad de la solución o su comportamiento en situaciones variadas o imprevistas, centrándose los esfuerzos en realizar una tarea concreta y estructurando el entorno si es preciso. Si bien este enfoque ha tenido un gran éxito en aplicaciones industriales, no proporciona un buen punto de partida para la creación de un modelo de visión aplicable a sistemas inteligentes que han de desenvolverse en entornos variables.

El modelo de percepción visual que se presenta en este trabajo está básicamente encuadrado en un enfoque teórico, inspirándose en lo posible en observaciones empíricas pero rompiendo con ellas cuando éstas no ofrecen alternativas viables o cuando el conocimiento del fenómeno es vago.

1.2. Descripción del problema

Desde un enfoque empírico, la sensibilidad al movimiento es un aspecto fundamental de la percepción visual. Por un lado se analiza el movimiento de los ob-

jetos del campo visual para poder interactuar con ellos. Por otro, al moverse el observador dentro de entorno, se producen cambios en la imagen retiniana de los objetos que rodean al observador. En ambos casos se producen cambios espaciotemporales de luminiscencia en la imagen percibida y dichos cambios son fuente de información sobre el entorno y el movimiento del observador respecto al entorno. El análisis de esa información permite establecer la existencia de los límites de detección del movimiento y las características diferenciales entre movimientos reales y los movimientos aparentes.

Ahora bien, existen evidencias que indican que en los sistemas de visión biológicos se utilizan dos mecanismos de medida del movimiento. Un mecanismo llamado de largo plazo en el que primero se realiza una búsqueda de características o rasgos significativos para la obtención de los vectores de desplazamiento y después se interpretan las correspondencias para poder realizar una estimación del movimiento en la escena. El segundo mecanismo, llamado de corto plazo, primero realiza el cálculo de las velocidades instantáneas para la obtención del **campo de velocidades o flujo óptico** y finalmente realiza una interpretación del campo de velocidades para poder realizar una estimación del movimiento. Estos dos mecanismos se ejecutan de manera concurrente lo que significa que son ejecutados en dos partes del cerebro diferentes pero simultáneamente.

Por otro lado el ojo humano, y en particular la fóvea, dispone de importantes características entre las que se destacan la máxima agudeza visual y la máxima sensibilidad al contraste. De tal manera que, el ojo humano, puede funcionar sobre un rango amplio de iluminación y también ser capaz de soportar diferentes cambios de intensidad en la iluminación.

Constantemente el ojo humano captura la información proyectada periódicamente en forma de imagen en la retina. La información es integrada de tal manera que los objetos de la escena aparecen estáticos o con movimientos suaves. Entonces, debido a que el tiempo empleado para percibir la información y procesarla es finito el sistema de visión es más sensible a los cambios percibidos en el momento de adquirir la nueva imagen.

En este trabajo se pretende abordar el análisis del movimiento mediante la obtención del flujo óptico guiada por los cambios. El análisis del movimiento es un campo de la visión artificial que ha tenido un importante auge en los últimos años. La razón es, en parte, al interés de sus aplicaciones y a que cada vez existen arquitecturas más aptas para realizar procesado en tiempo real. El análisis del movimiento

está relacionado, entre otros, con aplicaciones en tiempo real como la navegación, seguimiento y obtención de información sobre los objetos estáticos y en movimiento en una escena. Así mismo, el análisis del movimiento también es fundamental en problemas como la restauración de secuencias de imágenes, su compresión, obtención de imágenes y secuencias de imágenes de alta resolución a partir de secuencias de baja resolución, entre otras aplicaciones. De esta manera es posible deducir que al realizar un análisis de movimiento es inevitable que intervenga la variable de tiempo. El hecho es que la dimensión temporal es importante en el procesamiento visual fundamentalmente por dos razones: el movimiento aparente de los objetos en el plano de la imagen es fundamental para entender la estructura y el movimiento 3-D y los sistemas visuales biológicos utilizan el movimiento visual para extraer propiedades del mundo 3-D con poco conocimiento a priori sobre él.

De manera general, es posible decir que los problemas de análisis del movimiento se pueden resumir en 3 grupos básicos:

1. **Detección de movimiento.** El objetivo es detectar si hay movimiento en la escena. Tiene aplicaciones a seguridad y en problemas de compresión de vídeo.
2. **Detección y localización de objetos en movimiento.** Es más complicado que el anterior y puede incluir también la detección de trayectorias y predicción de futuras trayectorias.
3. **Obtención de propiedades 3-D de objetos a partir del movimiento.** Es un problema típico de Visión Artificial.

Se pretende crear un modelo para el análisis del movimiento obteniendo los vectores del flujo óptico donde la dirección indica hacia donde se mueve el objeto y el módulo la rapidez con la que se hace. El sistema visual deberá ser capaz de computar ambas cosas a la vez. Hasta este punto no hay nada nuevo, existen trabajos desarrollados que presentan distintas técnicas para calcular el flujo óptico y éstas se diferencian unas de otras por el tipo de restricción que utilizan. De hecho, de manera general todos los métodos requieren un pequeño intervalo temporal entre las imágenes consecutivas capturadas. La intención es evitar que ocurran grandes cambios entre dos imágenes consecutivas. Así pues, la obtención del flujo óptico desemboca en la determinación de la dirección del movimiento y de la velocidad del movimiento de todos los puntos de la imagen.

Desgraciadamente, los cambios en las intensidades de la imagen no se deben exclusivamente a los desplazamientos tridimensionales de las superficies representadas, sino que pueden obedecer a otros fenómenos, como por ejemplo, cambios de

iluminación en la escena. En estos casos aparece un *movimiento aparente* que podrá medirse, pero que no será debido a desplazamientos de los objetos. Existe una diferencia intrínseca entre el desplazamiento de los patrones de intensidad en la imagen y el campo de movimiento real, por lo que el flujo óptico no siempre es igual que el campo de movimiento.

Por lo antes dicho, el flujo óptico puede ser definido como el movimiento aparente de los niveles de intensidad de una imagen. Existen otras definiciones que asocian al flujo óptico a un conjunto denso de puntos de correspondencia entre imágenes sucesivas obteniendo, a partir de esas correspondencias, la velocidad de puntos o zonas de la imagen. El hecho es que, se defina como se defina, el flujo óptico es la mejor medida del movimiento, en el espacio bidimensional. Tan importante para un sistema artificial como para un organismo vivo. El único problema es que su cálculo no es una cuestión trivial. De hecho, el flujo óptico, sigue siendo un campo abierto en la investigación, dado que los sistemas de visión artificial todavía no lo han logrado calcular de forma óptima (precisión elevada en poco tiempo).

En la propuesta inicial para el cálculo del flujo óptico, se supone que la intensidad de la imagen en un punto dado se conserva en el tiempo. Esta consideración es comúnmente empleada en todos los sistemas de visión y asume que las variaciones de intensidad de la imagen se deben únicamente a los desplazamientos de los objetos, sin tener en cuenta los cambios de iluminación. Pero adicionalmente, también se considera una restricción de suavidad, suponiendo que el flujo óptico en un píxel es ligeramente diferente al flujo existente entre los píxeles vecinos. Todo esto es considerado bajo la condición de que la frecuencia de muestreo es alta, entre las imágenes consecutivas capturadas.

Ahora bien, uno de los mayores problemas a los que se enfrentan los sistemas de visión es la complejidad de los algoritmos de procesamiento de imágenes a utilizar, restricción inminente para que una plataforma o robot móvil pueda funcionar en **tiempo real**. Aunado a esto, también se tiene la gran cantidad de información a procesar, ya que cuanto más precisa se requiere ésta, más resolución deberán tener las imágenes a procesar. Otro problema, aunado al anterior, es el hecho de que en todos los sistemas de visión se realiza el **procesado de todos los píxeles de la imagen** o de la secuencia de imágenes, incluso si no existen cambios dentro de una secuencia de imágenes o de pares de imágenes consecutivas. Todo eso resulta muy costoso, para un procesado en tiempo real, y por demás innecesario puesto que muchas veces la mayoría de los píxeles, entre imágenes consecutivas, no cambian de valor. La razón de los pocos cambios es debido a la restricción impuesta en la cual

requieren un pequeño intervalo temporal entre las imágenes consecutivas capturadas y entre mayor sea la frecuencia de muestreo menos cambios existirán.

Estas restricciones de velocidad de procesado y la información masiva a procesar obligan a considerar ciertas alternativas, como la implementación de los algoritmos de sensorización visual con hardware hecho a la medida y la flexibilidad software, que implica una mayor lentitud en los algoritmos de visión artificial. Si se desea combinar ambas características para tener un módulo eficaz es necesario recurrir a tecnología reconfigurable, en este caso las FPGA's o CPLD's son la herramienta indicada para este propósito.

El presente proyecto nace en el grupo de investigación TAPEC, dentro del departamento de Informática de la Universidad de Valencia. El grupo de investigación ha realizado trabajos en el área de visión artificial, desde hace varios años, con énfasis en arquitecturas de procesamiento de imágenes basadas en lógica reconfigurable, entre las cuales están los sistemas visión log-polar y algunas otras aplicadas en plataformas móviles.

Con el avance de la tecnología, en particular la alta escala de integración en la fabricación de los circuitos integrados, se abre la posibilidad de implementar en un mismo chip todo un sistema de visión artificial, a un relativo bajo costo, con la utilización de una FPGA.

En este trabajo se plantea una nueva estrategia para el procesado de la información en tiempo real. Esta estrategia se denomina **procesado guiado por cambios**. Se eligió la ecuación iterativa de Horn para probar la efectividad de la técnica y por tratarse de una función ampliamente abordada. De esta manera, el cálculo de flujo óptico guiado por cambios pretende solucionar el problema de la gran cantidad de información a procesar. Por otro lado, con el uso de tecnología reconfigurable, en particular con las FPGA's, se pretende aumentar la velocidad de procesado mediante la realización de una arquitectura hecha a la medida, del tipo flujo de datos.

1.3. Objetivos

A partir de la descripción del problema, expuesta en la sección anterior, es posible concluir que el objetivo general del presente trabajo de investigación es:

Desarrollar una arquitectura hardware, utilizando dispositivos lógicos programables, para la implementación de un sistema de visión que realice un análisis de

movimiento de alta velocidad en tiempo real. El análisis del movimiento será hecho sólo para aquellos píxeles que cambiaron de intensidad.

Para la realización de este objetivo general surgen varios objetivos específicos, los cuales se pueden enumerar de la siguiente manera:

1. Estudio de los fundamentos fisiológicos y modelos para el análisis del movimiento.
2. Elección y estudio del algoritmo para el cálculo del flujo óptico.
3. Estudio de arquitecturas de procesamiento de imágenes.
4. Elección de una metodología de diseño correcta para realizar de forma eficiente el trabajo.
5. Análisis del procesado de imágenes guiado por cambios y su efecto en la obtención del flujo óptico.
6. Definición y diseño de la arquitectura que implementará los algoritmos establecidos.
   - Diseño del módulo de detección de cambios
   - Diseño del módulo de gradientes espacio-temporal
   - Diseño del módulo para el cálculo de velocidad
   - Diseño del módulo que implementa la restricción de suavidad.
   - Diseño del módulo de control
   - Diseño de módulos de memorias locales para etapas del sistema.
7. Diseño de etapas de interconexión, como la interfaz para el bus PCI y memoria local.
8. Simulación del sistema desarrollado.
9. Implementación física de la arquitectura propuesta.
10. Comprobación experimental del funcionamiento.

Se hace la aclaración de que en cada una de las etapas se realiza siempre y cuando sea cubierta con satisfacción la etapa anterior. Ya sea para la elección de un algoritmo específico o bien como paso necesario para avanzar a la siguiente fase del diseño.

1.4. Contenido

El presente trabajo se ha dividido en seis partes. Cada parte puede estar organizada en uno o varios capítulos que describen los diversos objetivos y fases del trabajo de investigación aquí desarrollado y la bibliografía utilizada. La primera parte describe el planteamiento del problema a resolver, recoge los fundamentos de este trabajo y el estado del arte del área. La segunda parte describe el diseño de la arquitectura propuesta para la realización del sistema de visión. La tercera parte consiste en implementar la arquitectura propuesta y llevar a cabo la experimentación requerida en la cual se verifica y comprueba el correcto funcionamiento del sistema. Aquí mismo se evalúa el sistema verificando el rendimiento y precisión del mismo. En la cuarta parte se recogen las conclusiones, aportaciones realizadas en el presente trabajo y trabajos a realizar en el futuro sobre esta línea de investigación. Las últimas dos partes, la quinta y sexta, presentan las citas bibliográficas y apéndices respectivamente. En los apéndices se incluye el código VHDL de los módulos diseñados para el sistema y una breve descripción del bus PCI.

Cada parte a su vez está organizada en capítulos que describen los diversos objetivos específicos y etapas del trabajo de investigación aquí desarrollado.

En el **capítulo 1** se realiza una breve introducción y descripción general del problema a resolver, así como los objetivos y el contenido del trabajo de investigación.

En el **capítulo 2** se realiza un estudio de diferentes modelos para el análisis del movimiento en el cual se decide por un algoritmo y técnica específica a utilizar.

En el **capítulo 3** se presenta un estudio de diferentes arquitecturas utilizadas para el procesamiento de imágenes. También se revisan arquitecturas existentes que estudian y realizan el cálculo del flujo óptico.

En el **capítulo 4** se expone el principio del procesado de imágenes guiado por cambios. Aquí se hace una evaluación por software de la técnica propuesta encontrando información y consideraciones para su implementación en hardware.

En el **capítulo 5** realiza el desarrollo de cada uno de los módulos que componen el sistema.

En el **capítulo 6** se presentan las simulaciones funcionales y digitales de los bloques que conforman al sistema. También es posible decir que incluye la parte final del diseño del sistema propuesto. Esto es debido a que en función de los resultados obtenidos de las simulaciones, se seleccionó uno u otro componente para el diseño, en

función del rendimiento y los recursos hardware necesarios para la implementación. En este mismo capítulo se realiza el procedimiento de la compilación y síntesis del diseño.

En el **capítulo 7** se realiza la implementación y verificación del sistema diseñado. Se expone como se programa la FPGA utilizada, el desarrollo de un programa para tomar el control del sistema y el diseño del controlador necesario para acceder al bus PCI.

En el **capítulo 8** se desarrollan los experimentos necesarios, para procesar las imágenes mediante la técnica aquí propuesta. De esta manera se calcula el flujo óptico guiado por cambios evaluando el rendimiento y la precisión del sistema.

En el **capítulo 9** se recogen las conclusiones, aportaciones realizadas en el presente trabajo y trabajos a realizar en el futuro.

Por último, se presentan las referencias bibliográficas y apéndices. Las referencias bibliográficas incluyen aquellas a las que hace referencia el trabajo y las publicaciones que ya se han realizado del presente trabajo de investigación. Los apéndices contienen el código VHDL de los módulos diseñados para el sistema y una breve descripción del bus PCI.

Capítulo 2

Análisis del movimiento

2.1. Introducción: Detección del movimiento

El sistema de visión humano percibe el movimiento con muy poco esfuerzo. Durante la percepción del movimiento, el sistema de visión procesa escenas dinámicas de forma natural, identifica sin dificultad los objetos que componen una escena y detecta el comportamiento temporal de objetos que participan en la escena.

Sin lugar a duda la percepción visual es el sistema sensorial más importante del ser humano, proporciona más de tres cuartas partes de la información que procesa el cerebro. La percepción visual ayuda en la estimación del movimiento relativo entre objetos, estimación del movimiento propio, procesos de orientación, discriminación entre distintos objetos e incluso en procesos de reconocimiento, ya sea para seguir algún objeto dentro de una escena o para ubicar objetos particulares en la misma. En resumen, la finalidad de la percepción visual consiste en transformar las percepciones visuales en conocimiento, permitiendo al sistema tomar decisiones a partir del entorno en el que se encuentra inmerso. Por lo tanto, para cualquier sistema de visión, es indispensable efectuar un análisis del movimiento de los objetos dentro de una escena.

El análisis de movimiento en imágenes es un área que está creciendo en numerosas aplicaciones tales como: codificación de las imágenes de vídeo mediante compensación [WSG05], robótica móvil [LdTRF06], tratamiento de las imágenes de satélite [SS04], seguimiento de objetos [SV05] [KP06] [MJE07], navegación autónoma [KLW05], tratamiento de imágenes biológicas y médicas [Bou03] [Aya03], vigilancia y su supervisión [BNS+06], realidad virtual e interfaces [CNBZ05], restauración de

imágenes [TCP04] y visión activa [CKC05].

Sin embargo, en cualquier tipo de aplicación, el problema en la detección del movimiento es particularmente interesante cuando el objetivo que se persigue es el de la localización espacial de los objetos móviles dentro de una escena. Esta detección de movimiento siempre está fuertemente ligada a la detección de cambios en la imagen. Cuando existen objetos en movimiento en una escena, siempre habrá cambios en la intensidad de los píxeles de la imagen. Este hecho ha dado lugar a una extensa investigación en el área de análisis del movimiento. No obstante, desde un punto de vista práctico, se puede decir que existen tres grandes grupos de problemas relacionados con el movimiento [Paj01]:

1. La detección del movimiento. Consiste sólo en registrar cualquier movimiento detectado en una escena.
2. La detección y localización de los objetos en movimiento. Se refiere a la detección del objeto, conocer la trayectoria de su movimiento y la predicción de la futura trayectoria. Todo eso considerando la posibilidad de que la cámara esté estática y los objetos se muevan en la escena, o la cámara se mueva y los objetos estén estáticos o bien tanto la cámara y los objetos estén en movimiento simultáneamente.
3. La obtención de las propiedades 3D de los objetos. Estas propiedades se obtienen mediante el uso de un conjunto de proyecciones 2D adquiridas en distintos instantes de tiempo del movimiento de los objetos.

Con frecuencia el análisis de escenas dinámicas de una secuencia de imágenes se denomina *análisis del movimiento* o *análisis dinámico de imágenes* [Paj01]. Comúnmente un sistema de análisis del movimiento está basado en un cierto número de imágenes consecutivas, dos o más dentro de una secuencia. Este hecho puede ser considerado como un proceso de análisis entre imágenes estáticas.

Es natural suponer que una secuencia de imágenes ofrece más información con respecto a un sistema estático. Pero también es evidente que, el considerar una secuencia se imágenes consecutivas, aumentará significativamente la cantidad de datos a procesar. Esto hace que la cantidad masiva de datos a procesar sea el principal problema de un sistema de visión.

Por otro lado, la necesidad de trabajar con una secuencia de imágenes hace inevitable introducir la variable tiempo t. La entrada, a un sistema de análisis del movimiento, es representada por $f(x, y, t)$, donde x e y son las coordenadas espa-

ciales en la imagen de la escena en un instante de tiempo t. El valor de la función $f(x, y, t)$ representa la intensidad del pixel (x, y) en la imagen t, representado comúnmente por $I(x, y, t)$. Sin embargo, no todo es tan fácil como la incorporación de una nueva variable.

Anteriormente se habló de problemas a solucionar en el área del análisis del movimiento desde un punto de vista práctico. Ahora bien, desde un punto de vista teórico existen problemas que se deben tener en cuenta. Estos problemas consisten en efectos que surgen involuntariamente en el sistema de visión, que puede ser desde el tipo de modelo considerado para la formación de la imagen hasta fenómenos que incluso el ser humano es incapaz de detectar. Algunos de estos problemas son [Cha01]:

1. **Modelo de formación de la imagen.** Esto se refiere a qué instrumento será usado como observador. Ya sea el modelo de perspectiva utilizado, aunque mayoritariamente el modelo usado es el de "*pinhole*", e incluso el tipo de sensor utilizado por la cámara.
2. **Regiones con poca información.** Muchas veces no existe suficiente información en el entorno próximo al objeto en estudio, como para determinar la existencia del movimiento. A este problema se le conoce como "*problema de apertura*".
3. **Cambios de intensidad sin la existencia del movimiento.** La mayoría de las aplicaciones reales, de sistemas de visión, consideran la restricción de que el brillo de la imagen a lo largo de la trayectoria del movimiento es constante, es decir, que cualquier cambio de la intensidad en un punto dado durante un tiempo es debido únicamente al movimiento.
4. **Solapamiento espectral.** Este problema está asociado al muestreo temporal propio de una secuencia de imágenes. En el dominio frecuencial, el solapamiento espectral o "*aliasing*" provocará, para una señal, una repetición del espectro de dicha señal.
5. **Movimientos multiples.** Muchas veces objetos en movimiento sufren rotaciones, expansiones o contracciones. Una alternativa comúnmente utilizada por muchos modelos, es asumir que el movimiento en una determinada región se puede expresar en base a traslaciones. Bajo ese principio, un único vector de velocidad se considera suficiente para describir el movimiento en esa región.

Con la finalidad de solucionar los problemas antes mencionados se han prop-

uesto múltiples algoritmos de estimación del movimiento [TJN98]. La mayoría de estos algoritmos surgen, como es común en los sistemas artificiales, de estudiar los sistemas de visión de animales avanzados [Gib86], [Gle02]. Dichos estudios proporcionan información muy importante ayudando a nuevas propuestas de algoritmos para el análisis de movimiento [MRD+06], [Bol00], [RJK06], [ZAS06], [KMK06].

Estos estudios proporcionan información muy importante lo que ayuda al desarrollo de posibles soluciones a dichos problemas [Gib86], [Gle02].

Existen indicios de que los sistemas de visión biológicos utilizan dos mecanismos de medida del movimiento [Váz96]. Un mecanismo llamado de largo plazo en el que primero se realiza una búsqueda de características o rasgos significativos para la obtención de los vectores de desplazamiento y después se interpretan las correspondencias para poder realizar una estimación del movimiento en la escena. El segundo mecanismo, llamado de corto plazo, primero realiza el cálculo de las velocidades instantáneas para la obtención del campo de velocidades o flujo óptico y finalmente realiza una interpretación del campo de velocidades para poder realizar una estimación del movimiento. Estos dos mecanismos se ejecutan de manera concurrente como lo prueban recientes estudios biológicos.

Existen diversas formas para clasificar a los algoritmos de detección del movimiento. Una primera clasificación podría ser entre las técnicas de detección del movimiento 3D y las técnicas de detección del movimiento 2D. El interés de este trabajo está enfocado en la detección de movimiento 2D por tal razón sólo se van a abordar las técnicas o métodos desarrollados para la estimación del movimiento 2D. La mayoría de los algoritmos, de estimación del movimiento 2D, pueden ser clasificados en tres grandes bloques: los métodos basados en la obtención del flujo óptico, los método basados en la correspondencia y los métodos diferenciales. Dentro de cada técnica existen distintos métodos que a su vez, algunos de ellos, pueden sub-clasificarse, tal como se muestra en la figura 2.1.

En este capítulo se presentará la formulación analítica del movimiento denominado *campo de movimiento* o *campo de velocidad*, que puede considerarse como la representación bidimensional (2D) de un movimiento tridimensional (3D). En la representación 2D cada punto tiene asignado un vector velocidad correspondiente a la dirección del movimiento, la velocidad y la distancia respecto un observador. También se presenta un enfoque diferente que analiza el movimiento a partir de la obtención del flujo óptico, que requiere un pequeño intervalo temporal entre imágenes consecutivas y que no ocurran grandes cambios entre dos imágenes consecutivas.

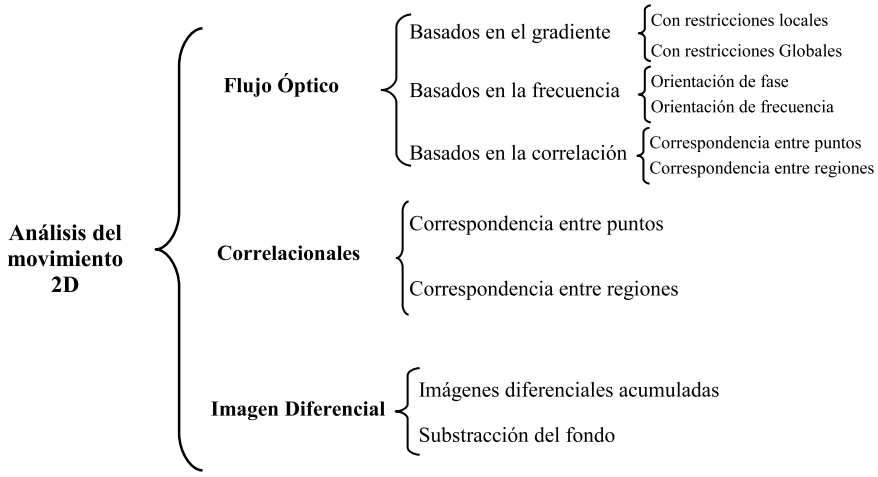


Figura 2.1: *Clasificación de las distintas técnicas para el análisis del movimiento.*

Además se presenta el método de correspondencia de regiones, donde la diferencia del desplazamiento entre las imágenes se minimiza sobre un conjunto de regiones locales empleando algún mecanismo de búsqueda apropiado. Finalmente se presenta el método diferencial que se basa en la obtención de la imagen diferencia, que se obtiene mediante la sustracción de los niveles de grises en dos imágenes consecutivas de la secuencia.

2.1.1. Campo de movimiento

El campo de movimiento es el campo de velocidades 2D de los puntos de la imagen, inducido por el movimiento relativo entre la cámara y la escena [Paj01]. En el campo de movimiento cada punto tiene asignado un *vector velocidad* correspondiente a la dirección del movimiento, la velocidad y la distancia respecto un observador. Así, se puede definir que el campo de movimiento es el desplazamiento inducido en los píxeles de la imagen por el movimiento relativo de los objetos de la escena, como se muestra en la figura 2.2. En este apartado se presentará la formulación analítica del movimiento. Existen distintas formulaciones para realizar la proyección, como son: la ortográfica, la perspectiva esférica, la perspectiva débil y la perspectiva plana (*pinhole*). Una revisión detallada sobre las proyecciones se puede encontrar en [HZ04] y [FJ03].

En este trabajo se realizará la formulación analítica utilizando la perspectiva plana por su fácil implementación. Se toma un sistema de coordenadas, ligado a la cámara, con el origen en el centro de la proyección perspectiva y el eje z ortogonal

al plano de la imagen, como se aprecia en la figura 2.2.

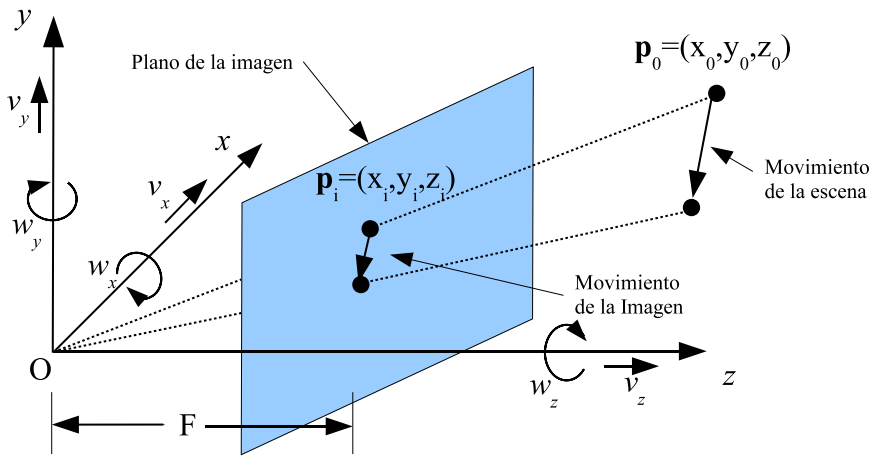


Figura 2.2: *Geometría para el análisis del movimiento en el plano de la imagen y el movimiento inducido en un píxel.*

Si la cámara se moviera con relación al objeto con una velocidad de traslación v y de rotación w sobre su origen, es posible decir que el objeto se mueve respecto a la cámara a una velocidad de traslación v y de rotación w, pero en sentido contrario $(-v, -w)$. Entonces la velocidad relativa instantánea de un punto $\mathbf{p}$ en la escena está dada por $-(v + w \times p)$, donde $\mathbf{p}$ es el vector de posición con coordenadas (x_0, y_0, z_0). Se pueden expresar las componentes de la velocidad instantánea de dicho punto como:

$$\begin{aligned} \dot{x}_0 &= -v_x - w_y z_0 + w_z y_0, \\ \dot{y}_0 &= -v_y - w_z x_0 + w_x z_0, \\ \dot{z}_0 &= -v_z - w_x y_0 + w_y x_0, \end{aligned} \tag{2.1}$$

Ahora bien, suponiendo que existe sólo un movimiento de traslación, se tiene:

$$\begin{aligned} \dot{x}_0 &= -v_x, \\ \dot{y}_0 &= -v_y, \\ \dot{z}_0 &= -v_z, \end{aligned} \tag{2.2}$$

Aplicando la transformación perspectiva plana [HZ04], las coordenadas de la imagen del punto **p** están dadas por:

$$x_i = \frac{Fx_0}{z_0} \quad e \quad y_i = \frac{Fy_0}{z_0} \tag{2.3}$$

donde F es la longitud focal. Derivando con respecto al tiempo y sustituyendo se obtiene la velocidad (de traslación) de la imagen que viene dada por las expresiones:

$$\dot{x}_i = \frac{-Fv_x + x_i v_z}{z_0} \quad e \quad \dot{y}_i = \frac{-Fv_y + y_i v_z}{z_0} \tag{2.4}$$

En realidad, el campo de velocidad de la imagen $[\dot{x}, \dot{y}]$ se expresa como la suma de un campo de traslación y un campo de rotación [Paj01]. En este caso como no se consideró la existencia de un movimiento de rotación, sólo se obtuvo la componente de traslación.

2.2. Flujo óptico

Existen distintas técnicas para calcular el flujo óptico y éstas se diferencian unas de otras por el tipo de restricción que utilicen para el cálculo. De manera general todos estos métodos requieren un pequeño intervalo temporal entre imágenes consecutivas. La intención es evitar que ocurran grandes cambios entre dos imágenes consecutivas. La obtención del flujo óptico desemboca en la determinación de la dirección del movimiento y de la velocidad del movimiento de todos los puntos de la imagen. El objetivo inmediato del análisis de imágenes basado en el flujo óptico es determinar el campo del movimiento.

Desgraciadamente, los cambios en las intensidades de la imagen no se deben exclusivamente a los desplazamientos tridimensionales de las superficies representadas, sino que pueden obedecer a otros fenómenos. Por ejemplo, los cambios en la iluminación de la escena [TLCH05] [AC06]. En estos casos aparece un *movimiento aparente* que podrá medirse, pero que no será debido a desplazamientos de los objetos. Existe una diferencia intrínseca entre el desplazamiento de los patrones de intensidad en la imagen y el campo de movimiento real, por lo que el flujo óptico no siempre es igual que el campo de movimiento. Como ejemplo, una esfera lisa de color uniforme que gira sobre su eje; aunque los puntos de su superficie se desplacen como se ve en la figura 2.3(a), no se va a percibir ningún movimiento o cambio en

los patrones de intensidad, aún cuando está rotando la esfera. Si por el contrario la misma esfera permanece estática mientras la fuente de iluminación varía su posición, figura 2.3(b), se producirá un desplazamiento de los patrones de intensidad en la imagen. Esto demuestra que es relativamente fácil que la diferencia entre flujo óptico y campo de movimiento se manifieste.

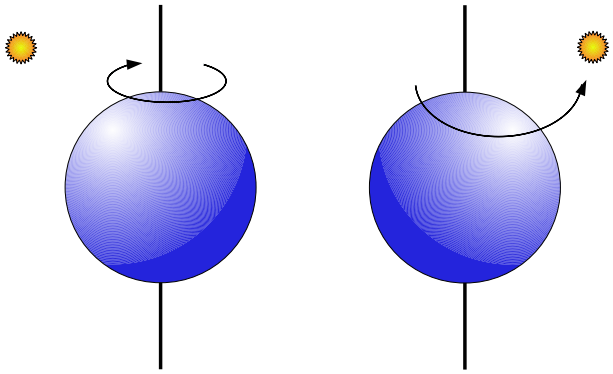


Figura 2.3: *Ejemplo donde el flujo óptico no corresponde al campo del movimiento. a) La esfera gira, pero no se producen cambios en la intensidad de los píxeles. b)La esfera permanece estática y la fuente de iluminación se desplaza, produciéndose un movimiento aparente.*

Por lo antes dicho, el flujo óptico puede ser definido como el movimiento aparente de los niveles de intensidad de una imagen. Para determinar este movimiento es necesario establecer algún tipo de modelo que permita relacionar las intensidades de los píxeles en la secuencia de las imágenes. Existen otras definiciones que asocian al flujo óptico a un conjunto denso de puntos de correspondencia entre imágenes sucesivas, obteniendo a partir de esas correspondencias, la velocidad de puntos o zonas de la imagen.

El hecho es que, se defina como se defina, el flujo óptico es la mejor medida del movimiento en el espacio bidimensional. El cálculo del flujo óptico es importante para un sistema artificial y para un organismo vivo [SKS01], [How05]. El único problema es que su cálculo no es una cuestión trivial. De hecho, el flujo óptico, sigue siendo un campo de investigación buscando nuevas estrategias y técnicas para su cálculo [XCS+06] [DRP+06] [DRM+06] [ALK07] [Tag07] [FSVG07]. La razón es que los sistemas de visión artificial todavía no han logrado calcularlo de forma óptima (precisión elevada en poco tiempo).

La propuesta inicial para el calculo del flujo óptico, fue realizada por Horn y Schunck en 1981 [HS81]. En dicho trabajo suponen que la intensidad de la imagen

en un punto dado se conserva en el tiempo. Esta consideración es comúnmente empleada en todos los sistemas de visión y asume que las variaciones de intensidad de la imagen se deben únicamente a los desplazamientos de los objetos, sin tener en cuenta los cambios de iluminación. Bajo dichas consideraciones, formalmente, para dos imágenes consecutivas separadas por un tiempo δt, si $I(x, y, t)$ es la intensidad de la imagen en el punto (x, y) y en el instante t, ha de cumplirse que:

$$I(x, y, t) = I(x + u\delta t, y + v\delta t, t + \delta t) \tag{2.5}$$

donde (u, v) es el vector de flujo óptico del píxel desconocido. El vector de flujo óptico depende de cada píxel considerado, siendo $u(x, y)$ la componente sobre el eje x y $v(x, y)$ la componente sobre el eje y.

Considerando que la intensidad luminosa varía de forma suave respecto de x, y, t, y desarrollando por series de Taylor se tiene:

$$\mathbf{I}(x, y, t) = \mathbf{I}(x, y, t) + \frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t + O^2 \tag{2.6}$$

donde O^2 es el conjunto de los términos de segundo orden y superiores. Despreciando O^2, cancelando $I(x, y, t)$ de ambos lados de la igualdad, dividiendo por δt y tomando el límite cuando δt tiende a cero, se obtiene la ecuación:

$$\frac{\partial I}{\partial x}\frac{dx}{dt} + \frac{\partial I}{\partial y}\frac{dy}{dt} + \frac{\partial I}{\partial t} = 0 \tag{2.7}$$

que puede ser representada, de forma más compacta, como:

$$I_x u + I_y v + I_t = 0 \tag{2.8}$$

donde I_x, I_y, I_t son las derivadas parciales de la imagen con respecto a x, y, t, respectivamente, y $\mathbf{v} = (u, v)$ representa el vector de flujo óptico en cada punto a determinar. Esta expresión es conocida como la "Ecuación de restricción de Flujo Óptico" (ERFO) [Gon99].

Se puede concluir que las restricciones básicas de la ERFO son:

1. La consideración de que un determinado patrón de brillo de la imagen permanezca constante. (Ecuación 2.5).
2. Considerar que los términos de segundo orden y superiores de la ecuación 2.6 sean despreciables, es decir, que el gradiente local de intensidad sea prácticamente lineal. Esto se puede asumir sí dx, dy, y dt son muy pequeños.

Retomando la ecuación de la ERFO, (2.8), se puede decir que I_x, I_y e I_t son datos que pueden ser obtenidos o al menos se pueden obtener mediante aproximaciones a partir de $I(x, y, t)$.

Una posibilidad comúnmente utilizada para obtener estos gradientes es mediante la aplicación de una mascara de 8 píxeles entre dos imágenes consecutivas, obtenidas en un tiempo k y $k+1$ [Hor86]. Como se presenta en la figura 2.4.

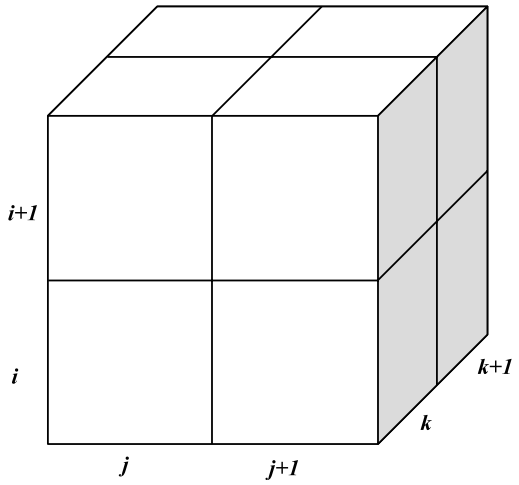


Figura 2.4: *Píxeles requeridos para calcular los gradientes espacio-temporal* (I_x, I_y, I_t).

Las ecuaciones para obtener los gradientes son representadas por:

$$I_x \approx \frac{1}{4}\{I_{i,j+1,k} - I_{i,j,k} + I_{i+1,j+1,k} - I_{i+1,j,k} + I_{i,j+1,k+1} - I_{i,j,k+1} + I_{i+1,j+1,k+1} - I_{i+1,j,k+1}\} \tag{2.9}$$

$$I_y \approx \frac{1}{4}\{I_{i+1,j,k} - I_{i,j,k} + I_{i+1,j+1,k} - I_{i,j+1,k} + I_{i+1,j,k+1} - I_{i,j,k+1} + I_{i+1,j+1,k+1} - I_{i,j+1,k+1}\} \tag{2.10}$$

$$I_t \approx \frac{1}{4}\{I_{i,j,k+1} - I_{i,j,k} + I_{i+1,j,k+1} - I_{i+1,j,k} + I_{i,j+1,k+1} - I_{i,j+1,k} + I_{i+1,j+1,k+1} - I_{i+1,j+1,k}\} \tag{2.11}$$

donde los subíndices *i,j,k* hacen referencia al píxel *(i,j)* en un instante *k*.

Ahora bien, una vez que se conocen los gradientes, el objetivo es obtener la velocidad $\mathbf{v} = (\frac{dx}{dt}, \frac{dy}{dt}) = (u, v)$. La ERFO proporciona una restricción para el cálculo del flujo óptico, pero por sí sola no puede determinar la velocidad, ya que es una ecuación con dos incógnitas (u y v). Como mucho, se puede obtener el módulo de

la velocidad normal al gradiente local (ver figura 2.5) correspondiendo éste a la distancia desde el origen a la recta descrita por la ecuación (2.8). Esta limitación es conocida como el problema de la apertura [BB95]. Gráficamente se muestra en la figura 2.6.

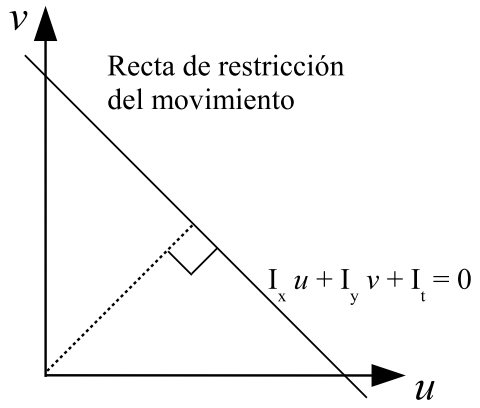


Figura 2.5: *Recta de restricción del flujo óptico usando el criterio del gradiente.*

La componente de la velocidad de la imagen en la dirección del gradiente (I_x, I_y) está dada por:

$$\mathbf{v}_\perp = \frac{I_t}{\sqrt{I_x^2 + I_y^2}} \tag{2.12}$$

El problema de la apertura consiste en que en una zona de la imagen sólo es posible calcular la componente del flujo óptico que está en la dirección del gradiente de la intensidad. No es posible medir la componente tangencial a dicho gradiente, a no ser que en la zona o apertura se encuentre una esquina donde la imagen presente suficiente *estructura* que ayude a determinar las dos componentes de un vector.

Sin embargo, como se ha dicho, dado que únicamente se dispone de una ecuación para estimar dos incógnitas, el cálculo de flujo óptico constituye un problema mal definido [BB95]. Por consiguiente es necesario añadir restricciones adicionales a la definición del problema para poder estimar valores más concretos. Esas restricciones, o bien relacionan de alguna manera los valores de flujo presentes en diferentes puntos de la imagen, imponiendo criterios más o menos globales, o bien tratan de añadir más ecuaciones a cada punto.

Existe una gran cantidad de algoritmos para calcular el flujo óptico y otros siguen apareciendo en la actualidad [Yea02], [BW05], [AA05], [KLW05]. Sin embargo, la gran mayoría de los algoritmos se pueden clasificar en tres grandes grupos [BBF94],

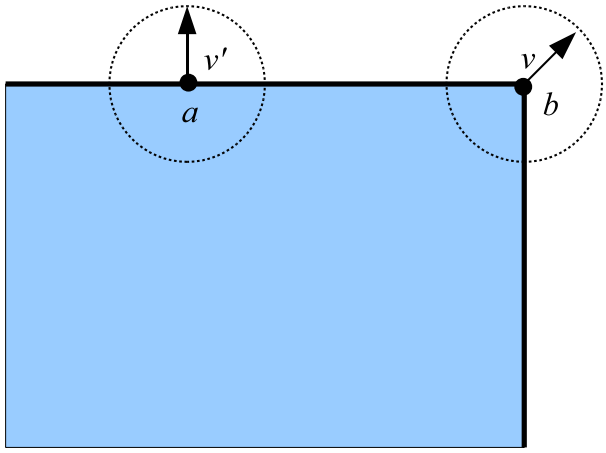


Figura 2.6: *Problema de la apertura. En el punto* $\boldsymbol{a}$*, dado su entorno inmediato, sólo se puede estimar el vector componente de* $\boldsymbol{v}$ *en la dirección del gradiente de la imagen. En* $\boldsymbol{b}$*, no existe esa ambigüedad debido a que se encuentra en una esquina y hace posible detectar un movimiento horizontal o vertical.*

[Bol00], [Cha01], [Luc03]:

1. **Métodos diferenciales.** También conocidos como métodos del gradiente. Estos métodos son basados en el gradiente espacio temporal.
2. **Métodos correlacionales.** Estos métodos son basados en la correspondencia de regiones, puntos o *características de interés* entre dos imágenes consecutivas.
3. **Métodos frecuenciales.** Estos métodos basan su criterio en la energía espacio temporal o espectrales.

Un estudio extenso y exhaustivo sobre las principales técnicas se pueden encontrar en [BBF95], [BB95] y complementarios a estos en [GMN+98].

2.2.1. Métodos diferenciales

Estos métodos fueron los primeros en formularse en [HS81]. Horn y Shunck utilizan los criterios basados en el gradiente espacio temporal, también llamados diferenciales, empleando de forma explícita la ERFO, ecuación 2.8. Sin embargo, como se dijo anteriormente, la ERFO no puede determinar la velocidad por si sola. Por tal razón se debe tener en cuenta algún tipo de restricción adicional. Así, dentro de estos métodos existen distintas clases de algoritmos en función del tipo de restricción

que utilizan.

Entre las distintas clases de este método basado en el gradiente, están aquellos que utilizan restricciones globales y aquellos que utilizan restricción locales. Los métodos con restricciones globales se caracterizan por que consideran toda la imagen en el cálculo. Estos utilizan normalmente un término de regularización de suavizado, como restricción adicional, para calcular un flujo óptico más denso en regiones más grandes de la imagen. Los métodos, con restricciones locales, son aquellos que sólo consideran una zona de la imagen. En ellas se utiliza la información de la velocidad normal en vecindades, para realizar una minimización por mínimos cuadrados que conduzca a la mejor elección del valor del flujo.

Métodos con restricciones globales

Los métodos globales utilizan explícitamente la ecuación de restricción de flujo óptico junto con un término de regularización, normalmente una restricción de suavidad.

El método con restricción global más estudiado, sin lugar a duda, es el implementado por Horn y Schunck. Incluso actualmente se continua trabajando sobre este método, en combinación con otros [BW05], como referencia para buscar nuevas estrategias en la obtención del flujo óptico [DDB04] o para buscar la convergencia del citado algoritmo [MM04]. En este método la restricción de suavidad se basa en suponer que el campo del movimiento varía suavemente en la mayor parte de la imagen. Esto significa que los vecinos de un píxel dado tendrán velocidades similares, involucrando los píxeles del resto de la imagen.

Existen distintas formas de expresar la restricción de suavidad [Hor86]. Una de ellas es minimizando el cuadrado de la magnitud del gradiente del flujo óptico. Otra es determinando la suma de los cuadrados de la Laplaciana de las componentes x e y del flujo óptico.

Horn y Schunck [HS81] proponen como medida de suavidad el cuadrado de la magnitud del gradiente e intenta minimizar la integral de dicha medida. Adicionalmente, dado que las medidas de la luminosidad de los píxeles en la imagen tendrán un error debido al error de cuantificación y al ruido, se intenta optimizar el error en la ERFO.

Básicamente su técnica se reduce a minimizar la cantidad del error al cuadrado.

$$E^2(x,y) = \int_D \int ((I_x u + I_y v + I_t)^2 + \lambda^2 (u_x^2 + u_y^2 + v_x^2 + v_y^2)^2)\, dx\, dy \qquad (2.13)$$

donde D es el dominio de interés y u_x^2, u_y^2, v_x^2 y v_y^2 son las derivadas parciales de las componentes de la velocidad al cuadrado. El primer término representa una solución a la ERFO, el segundo representa la restricción de suavidad global y λ^2 es un factor que pondera la importancia relativa entre los errores de ambos términos. Resolviendo la ecuación, utilizando cálculos de variaciones, el problema se reduce a dos ecuaciones diferenciales,

$$I_x^2 u + I_x I_y v = \lambda^2 \nabla^2 u - I_x I_t \quad y \quad I_x I_y u + I_y^2 v = \lambda^2 \nabla^2 v - I_y I_t \qquad (2.14)$$

donde ∇^2 es la Laplaciana. Una forma de calcular la Laplaciana de una matriz bidimensional, como es el caso de imágenes digitales, es mediante una máscara y realizando la convolución. Varias máscaras han sido propuestas en [Hor86]. Pero la que mejor resultado ha mostrado fue la propuesta por Horn y Schunck en [HS81], que se muestra en la figura 2.7.

1/12 *i-1,j-1*	1/6 *i-1,j*	1/12 *i-1,j+1*
1/6 *i,j-1*	-1 *i,j*	1/6 *i,j+1*
1/12 *i+1,j-1*	1/6 *i+1,j*	1/12 *i+1,j+1*

Figura 2.7: *Máscara utilizada para calcular la Laplaciana.*

Matemáticamente se puede representar como:

$$\nabla^2 u \approx \kappa(\overline{u} - u) \quad y \quad \nabla^2 v \approx \kappa(\overline{v} - v), \qquad (2.15)$$

donde κ es un factor de proporcionalidad igual a 3, $\overline{u}$ y $\overline{v}$ son los valores medios de la velocidad en las direcciones x e y en alguna vecindad de (x, y) y son definidas como:

$$\overline{u}_{i,j} = \frac{1}{6}(u_{i-1,j} + u_{i,j+1} + u_{i+1,j} + u_{i,j+1}) + \frac{1}{12}(u_{i-1,j-1} + u_{i-1,j+1} + u_{i+1,j+1} + u_{i+1,j-1}), \tag{2.16}$$

$$\overline{v}_{i,j} = \frac{1}{6}(v_{i-1,j} + v_{i,j+1} + v_{i+1,j} + v_{i,j+1}) + \frac{1}{12}(v_{i-1,j-1} + v_{i-1,j+1} + v_{i+1,j+1} + v_{i+1,j-1}), \tag{2.17}$$

Usando la aproximación de la Laplaciana, la ecuación (2.14) se puede definir como:

$$(\lambda^2 + I_x^2)u + I_x I_y v = (\lambda^2 \overline{u} - I_x I_t) \quad y \quad I_x I_y u + (\lambda^2 + I_y^2)v = (\lambda^2 \overline{v} - I_y I_t) \tag{2.18}$$

Estas ecuaciones pueden resolverse para u y v mediante:

$$u = \overline{u} - I_x \frac{I_x \overline{u} + I_y \overline{v} + I_t}{\lambda^2 + I_x^2 + I_y^2} \tag{2.19}$$

$$v = \overline{v} - I_y \frac{I_x \overline{u} + I_y \overline{v} + I_t}{\lambda^2 + I_x^2 + I_y^2} \tag{2.20}$$

Se puede ver que estas ecuaciones, (2.19) y (2.20), muestran una dependencia espacial en las velocidades. De tal manera que la determinación del flujo óptico está basada en un método iterativo utilizando pares de imágenes dinámicas consecutivas denominado de Gauss-Seidel y detallado en [Hor86].

Métodos con restricciones locales

Los métodos con restricciones locales parten de la ecuación de la ERFO, ecuación 2.8, y utilizan como restricción local la consideración de un flujo localmente constante. Uno de los métodos con restricciones locales que ha dado buen rendimiento, en cuanto a precisión y tolerancia al ruido, es el propuesto por Lucas y Kanade [GMN+98]. Una revisión exhaustiva de dicho método se puede encontrar en [BM04]. Su suposición principal es un modelo de velocidades constantes en una región espacial o ventana. Dentro de esta región se minimiza la siguiente expresión:

$$\sum_{(x,y)\in R} W^2(x,y)(\nabla I(x,y,t)\,\mathbf{v} + I_t(x,y,t))^2 \tag{2.21}$$

donde $W(x, y)$ es una función que otorga pesos a cada punto, favoreciendo a aquellos puntos situados más cerca del centro del entorno de la región y R es la región espacial o ventana de (x, y). La solución a esta ecuación está dada por:

$$A^T W^2 A\,\mathbf{v} = A^T W^2 \mathbf{b} \tag{2.22}$$

donde, para n puntos (x_i, y_i) pertenecientes a R en un instate t, A, W y $\mathbf{b}$ están definidas por:

$$\begin{aligned} A &= [\nabla I(x_1, y_1), ..., \nabla I(x_n, y_n)]^T \\ W &= diag[W(x_1, y_1), ..., W(x_n, y_n)] \\ \mathbf{b} &= -[I_t(x_1, y_1), ..., I_t(x_n, y_n)]^T \end{aligned} \tag{2.23}$$

Despejando $\mathbf{v}$ de la ecuación 2.22, se tiene:

$$\mathbf{v} = [A^T W^2 A]^{-1} A^T W^2 \mathbf{b} \tag{2.24}$$

que tiene solución cuando $A^T W^2 A$ es una matriz *no singular*. La matriz $A^T W^2 A$ es de dimensión 2×2 y viene dada por:

$$A^T W^2 A = \begin{bmatrix} \sum W^2(x,y) I_x^2(x,y) & \sum W^2(x,y) I_x(x,y) I_y(x,y) \\ \sum W^2(x,y) I_x(x,y) I_y(x,y) & \sum W^2(x,y) I_y^2(x,y) \end{bmatrix} \tag{2.25}$$

por otra parte, la matriz $A^T W^2 \mathbf{b}$ es de dimensión 2×1 y está dada por:

$$A^T W^2 \mathbf{b} = \begin{bmatrix} \sum W^2(x,y) I_x(x,y) I_t(x,y) \\ \sum W^2(x,y) I_y(x,y) I_t(x,y) \end{bmatrix} \tag{2.26}$$

Los métodos locales también sufren del problema de la apertura. Si la matriz $A^T W^2 A$ es singular cuando el gradiente es constante, en una o más direcciones, sobre la vecindad R, se puede interpretar como otro ejemplo del problema de la apertura. De tal manera que existen intentos para encontrar una solución analítica a dicho problema [Nag90]. También se ha propuesto un enfoque probabilístico [Sim93] para dar solución a la ecuación 2.21.

2.2.2. Métodos basados en la correspondencia

Este grupo de métodos, estudiado ampliamente en [BB95] y complementado en [GMN+98], utilizan el criterio de seleccionar una serie de características de la imagen que posteriormente serán buscadas y emparejadas en imágenes consecutivas, para finalmente encontrar el flujo óptico. Una ventaja de este grupo de métodos es que son más eficientes, respecto a los métodos basados en el gradiente, cuando se dispone de muy pocos cuadros o cuando la relación señal-ruido de la imagen es demasiado pobre. Por otro lado, el campo de desplazamiento obtenido es menos denso que el que pudiésemos obtener utilizando el método del gradiente. Sin embargo, existen trabajos recientes, como [WB04], que buscan obtener una mayor densidad en los mapas de flujo óptico. Otro punto a considerar es su costo computacional, dado que requieren ejecutar algoritmos de selección de las características y su posterior detección de correspondencia.

Existen trabajos que presentan una mayor rapidez "relativa" de procesado, pero a costa de la perdida de precisión, como lo han probado en [LHH+98] y [GMN+98] y más recientemente en [OMTO02]. Sin embargo, este método para el procesamiento de imágenes ha sido desplazado por otros que presentan mejores resultados.

Este grupo de métodos también puede ser dividido en dos subgrupos: Los basados en la correspondencia de características y los basados en la correspondencia de regiones.

Métodos basados en la correspondencia entre características

Este método también se conoce como correspondencia entre puntos o puntos relevantes. Estas técnicas son empleadas habitualmente para estimar disparidad entre imágenes y recuperar el desplazamiento del observador [Fau93]. Las principales razones del uso de esta técnica son una menor presencia del problema de apertura y que el trabajo está enfocado a producir buenos resultados a partir de imágenes reales [RB07], la mayoría tomadas en el exterior.

Básicamente en este método se localizan características o rasgos, como por ejemplo, bordes o esquinas, y se busca el movimiento de éstas en una secuencia de imágenes. Básicamente este proceso se ejecuta en tres etapas. En la **primera etapa** se buscan o detectan características de referencia en dos o más imágenes consecutivas; si esta extracción se hace de forma correcta, se reduce la cantidad de información a ser procesada. El problema de esta etapa es que se requiere que las características se encuentren de una forma precisa y fiable, hecho que no es una tarea trivial,

dedicándose mucho trabajo computacional en los algoritmos de detección de características. En la **segunda etapa** se buscan las características entre la secuencia de imágenes. El problema de esta etapa, bien conocido de la correspondencia, se da cuando se realizan búsquedas con un potencial ambiguo; a no ser que se conozca que el desplazamiento de la imagen va a ser menor que la distancia entre características, se debe encontrar algún método para elegir entre distintas coincidencias potenciales. En la **tercera etapa** se calcula el campo de flujo óptico.

El caso más simple en este método ocurre cuando se utilizan dos imágenes consecutivas y dos conjuntos de características para producir un conjunto único de vectores de movimiento. De un modo alternativo, las características de una imagen pueden ser usadas como puntos de origen para el uso de otros métodos (como ejemplo los métodos basados en el gradiente) para a continuación encontrar el flujo óptico.

Las características comúnmente utilizadas para el emparejamiento en este método son los bordes [II03], [TM02] y las esquinas [SSCW98]. Sin embargo, su costo computacional es muy elevado, pues se requieren ejecutar algoritmos de selección de las características, algoritmos de detección de correspondencia para finalmente estimar el campo de flujo óptico. Este proceso hace que estos algoritmos sean inconvenientes, por su costo computacional, para aplicaciones en tiempo real.

Métodos basados en la correspondencia entre regiones

De manera general en estos métodos se considera que se conserva la distribución de intensidad de la región que rodea al punto donde se desea evaluar el movimiento. Por lo tanto, para cada punto donde se desea medir el flujo óptico en un instante determinado, se crea una ventana de puntos que rodean al mismo (llamada ventana de referencia) y se busca la máxima correspondencia de la misma dentro de un conjunto de ventanas de igual tamaño y centradas en los puntos contenidos en una ventana de posibles candidatos (llamada ventana de búsqueda).

En este grupo de métodos también existen varias clases. La distinción entre cada una de las clases radica en el tipo de criterio de error de correspondencia. Un criterio de error que ha tenido un relativo éxito utiliza operadores basados en la suma de diferencias al cuadrado (SSD). Dadas dos imágenes I_1 e I_2, se define la medida de correspondencia como:

$$SSD_{1,2}(\mathbf{x}, \mathbf{d}) = \sum_i \sum_j W(i,j) \cdot [I_1(x+i, y+j) - I_2(x+i+d_x, y+j+d_y)]^2 \quad (2.27)$$

donde $W(i,j)$ es una función de ventana, $\mathbf{x} = (x,y)$ son las coordenadas de cada punto de la imagen y el desplazamiento $\mathbf{d} = (d_x, d_y)$ sólo puede tomar valores enteros.

En Anandan se emplea una estrategia de emparejamiento dentro de una pirámide Laplaciana basada en SSD [Ana89]. También se agrega a la minimización una medida de distancia dando una mayor ponderación o peso a los desplazamientos pequeños que a los grandes. Así, son calculados los desplazamientos sub-píxel al minimizar la función de error SSD. Adicionalmente esta técnica emplea una restricción de suavidad para la estimación del flujo óptico realizando los cálculos de forma iterativa hasta alcanzar la precisión sub-píxel deseada. Anandan calcula la velocidad utilizando 10 iteraciones con $k_1 = 150, k_2 = 1$ y $k_3 = 0$ mediante la función [Ana89]:

$$\mathbf{v}^{k+1} = \overline{\mathbf{v}}^k + \frac{c_{max}}{c_{max}+1}[(\mathbf{v}_0 - \overline{\mathbf{v}}^k)\cdot \mathbf{e}_{max}]\mathbf{e}_{max} + \frac{c_{min}}{c_{min}+1}[(\mathbf{v}_0 - \overline{\mathbf{v}}^k)\cdot \mathbf{e}_{min}]\mathbf{e}_{min} \quad (2.28)$$

donde c_{max} y c_{min} son las medidas de confianza máxima y mínima, $\mathbf{e}_{max}$ y $\mathbf{e}_{min}$ son la curvatura de dirección máxima y mínima, respectivamente, de la superficie de la SSD mínima, $\mathbf{v}_0$ denota el desplazamiento en el nivel superior de la pirámide y $\overline{\mathbf{v}}^k$ es el promedio de la velocidad $\mathbf{v}^k$, que es calculada por la mascara,

$$\frac{1}{4}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad (2.29)$$

donde inicialmente $\overline{\mathbf{v}}^0$ toma el valor de $\mathbf{v}_0$. Se puede encontrar información más detallada sobre este trabajo en [Ana89].

Singh también utiliza la función de error SSD [Sin90]. Primero calcula la función SSD sobre tres imágenes consecutivas I_{-1}, I_0 e I_{+1} o sea sobre dos pares de imágenes consecutivas.

$$SSD_0(\mathbf{x},\mathbf{d}) = SSD_{0,+1}(\mathbf{x},\mathbf{d}) + SSD_{0,-1}(\mathbf{x},\mathbf{\text{-}d}), \quad (2.30)$$

Después SSD_0 se convierte en una distribución de probabilidad, para finalmente calcular un velocidad sub-píxel $\mathbf{v}$=(u_x, v_y) como la media de esa distribución, ponderada por los desplazamientos $\mathbf{d}$, mediante:

$$u(x) = \frac{\sum R(\mathbf{x},\mathbf{\text{-}d})d_x}{\sum R(\mathbf{x},\mathbf{\text{-}d})}, \quad y \quad v(y) = \frac{\sum R(\mathbf{x},\mathbf{\text{-}d})d_y}{\sum R(\mathbf{x},\mathbf{\text{-}d})} \quad (2.31)$$

donde R($\mathbf{x}$,$\mathbf{d}$) está dada por:

$$R(\mathbf{x},\mathbf{d}) = exp[\frac{ln(0{,}95)SSD_0(\mathbf{x},\mathbf{d})}{min(SSD_0(\mathbf{x},\mathbf{d}))}] \quad (2.32)$$

Finalmente Singh sugiere emplear este procedimiento dentro de un esquema jerárquico utilizando pirámides Laplacianas.

Esta clase de métodos proporcionan un campo de flujo óptico poco denso además de que presentan dos puntos críticos para su implementación. El primero de ellos es la carga computacional. El segundo punto, que afecta también al primero, es la elección adecuada de la ventana para realizar la correlación. Cuanto mayor sea, más rápido convergerá la función de error (con menos pasos), pero con una elevada carga computacional en cada paso.

Para solucionar los puntos críticos anteriores, Camus propone una novedosa estrategia, en [Cam97]. En dicho trabajo se plantea la incorporación de dos modificaciones para disminuir el tiempo empleado en la búsqueda de la correlación. La primera modificación implica el uso de un tiempo de muestreo variable de la imagen, que permite intercambiar el espacio con el tiempo. La segunda modificación permite extender la aplicación del algoritmo para crear un campo de flujo con múltiples velocidades, que permita transformar las búsquedas espaciales cuadráticas en otras lineales en el tiempo. Además asume que el máximo desplazamiento posible de un píxel de la imagen en cualquier dirección, está limitado a un valor. El valor que se tome para esta magnitud dependerá de la velocidad esperada en el movimiento de la imagen observada.

Basándose en las modificaciones anteriores este algoritmo realiza una búsqueda exhaustiva sobre la frontera cercana, a través de n imágenes. Para cada píxel se detecta, mediante una función error, el valor con la correspondencia más óptima. Posteriormente el flujo óptico es calculado como $\frac{1}{\delta t}(\delta x, \delta y)$. En [GMN+98] se implementan dos versiones de este algoritmo. El primero utiliza 3 imágenes y el segundo utiliza 15 imágenes. Como es lógico de esperar, el primer trabajo se ejecutan en un menor tiempo pero es más sensible al ruido.

Otro trabajo, en el cual se tiene como finalidad generar un campo de flujo óptico denso, es el desarrollado por Proesmans et al. en [PGPO94]. Allí se propone un algoritmo que utiliza un método similar al de Horn y Schunck. Solo que además incorpora un mecanismo de correlación dentro de las ecuaciones de restricción. De tal forma que para cada punto (x, y) de la primera imagen, se calcula el flujo óptico (u, v) que puede ser utilizado para encontrar su punto correspondiente $(x+u, y+v)$ en la segunda imagen.

2.2.3. Métodos frecuenciales

Estos métodos consideran que de una secuencia de imágenes es posible analizar el movimiento en el dominio de las frecuencias espacio temporal (dominio de Fourier). De tal manera que estos métodos se caracterizan por utilizar filtros espacio-temporales. En base a la interpretación temporal del movimiento, analizan el espectro frecuencial considerando diversas orientaciones según las velocidades y orientaciones del movimiento.

Dado que estos métodos requieren un soporte temporal considerable para un correcto funcionamiento dificultan a *priori* su empleo en aplicaciones que requieren ejecutarse en tiempo real. Por esta razón sólo se destacarán características y clases dentro de esta familia de métodos.

Se debe destacar que con estos métodos se puede detectar el movimiento de conjuntos de puntos con formas aleatorias, que sería muy difícil con métodos de correspondencia o con métodos basados en el gradiente. En el espacio de Fourier la energía orientada producida por el filtrado puede permitir la detección del movimiento. Una revisión reciente para el análisis de movimiento en el dominio de la frecuencia se presenta en el trabajo desarrollado por Ahuja [AB06].

Básicamente estos métodos se pueden clasificar dentro de aquellos que utilizan filtrado sensible a la orientación y los que utilizan el filtrado sensible a la fase.

Métodos basados en el filtrado sensible a la orientación

En [FA91] se presenta el diseño y uso de filtros sensibles a orientaciones arbitrarias. Para estos propósitos son comúnmente utilizados los filtros de Gabor, que son filtros creados mediante la multiplicación de funciones Gaussianas espacio-temporales con funciones trigonométricas, para conseguir una limitación en banda y una selección en los dominios espaciales y frecuenciales. Los filtros ortogonales de Gabor se usan para encontrar la energía de Gabor en 12 orientaciones diferentes y en varias frecuencias espaciales diferentes.

En [Jah90] y [Jah94] se realizan estudios analíticos de procesos de bajo nivel en la imagen, a partir de técnicas de filtrado sensibles a la orientación.

Métodos basados en el filtrado sensible a la fase

Dentro de las técnicas que utilizan un filtrado sensible a la fase se tiene el trabajo de Fleet y Jepson. En [LHH$^+$98] se compara la precisión y eficiencia de dicho

algoritmo. El algoritmo se presenta como el que tiene menor error. Sin embargo, es el que más tiempo requiere para su procesado. En el algoritmo, desarrollado en [FJ90], se realiza la extracción de la información local, contenida en la fase de la imagen para definir al movimiento, basándose en los movimientos de los contornos de la fase. Se utilizan filtros de Gabor, paso-banda, y son sintonizados para detectar distintas velocidades.

Guatama desarrolla un algoritmo basado en la fase [GVH02]. En dicho algoritmo se realiza un filtrado espacial a la secuencia de imágenes y se calcula las componentes de velocidad mediante una red neuronal recurrente.

Otras técnicas como [Mye03] combinan el método basado en la fase con un operador detector de características.

2.3. Análisis del movimiento basado en la correspondencia

La detección del movimiento basado en la correspondencia o en la correlación funciona muy bien aun para intervalos de muestreo relativamente altos o para grandes desplazamientos entre dos imágenes, algo que no sucede con el método de flujo óptico. La razón es que al emplear el método del flujo óptico existe la restricción de que sólo se puede utilizar si la diferencia entre las dos imágenes consecutivas es pequeña.

Estos métodos se basan en la búsqueda de los vectores de desplazamiento entre la imagen de referencia y la imagen actual, bajo un criterio de similitud. Existen diferentes criterios de similitud [TJN98]. Un criterio comúnmente utilizado para encontrar el vector desplazamiento $C(i,j)$, consiste en minimizar la función objetivo que calcula el valor absoluto de la diferencia entre la ventana de la imagen de referencia I_{k-1} con una ventana de la imagen actual I_k, y es representada por la ecuación:

$$C(i,j) = \sum_{(x,y)\in\Re} \mid I_k(x,y) - I_{k-1}(x+i, y+j) \mid^p \tag{2.33}$$

donde $\Re$ es una región de la imagen I_k que es comparada con la imagen I_{k-l} y p es la potencia del valor absoluto. Cuando $p = 1$ se considera el criterio del Error Absoluto Medio (MAE) y cuando $p = 2$ se considera el Error Cuadrático Medio (MSE).

El estimador mínimo del criterio de error se define como:

$$\hat{i}, \hat{j} = arg\, min_{(i,j)}\, C(i,j) \tag{2.34}$$

Idealmente la región de búsqueda sería toda la imagen pero esto no es práctico, tanto por el tiempo necesario para realizar la correlación en toda la imagen como por la frecuencia de muestreo utilizada. Por tal razón la región se restringe a un área de búsqueda alrededor de la localización del píxel, característica o región de encaje sobre la imagen actual.

Esta técnica de detección del movimiento se puede dividir en dos clases, considerando el método de correspondencia utilizado. Entre estas clases se tiene: correspondencia entre características o puntos de interés y correspondencia entre regiones. En [TJN98] se hace una revisión de los métodos de estimación de movimiento, para la compresión de video, donde se agrega una clase llamada correlación de fase.

2.3.1. Métodos basados en la correspondencia de características o puntos de interés

En primer lugar, este método encuentra los puntos significativos o características en todas las imágenes de la secuencia (bordes, esquinas, líneas, etc.). Después de la búsqueda y detección de las características o de los puntos de interés se aplica un procedimiento de emparejamiento, que busca correspondencias entre dichos puntos de la secuencia de imágenes. Finalmente, el resultado es la construcción de un campo de velocidad más o menos denso dependiendo del número de puntos de interés y de las características del algoritmo.

Suponiendo que los puntos de interés han sido localizados, sobre una secuencia de imágenes, se hace necesario obtener una correspondencia entre los puntos de interés basándose en una medida de similitud. La medida de similitud es una medida cuantitativa del mayor o menor grado que indica que entidades se corresponden entre sí. Hay distintas funciones que nos permiten medir esto, como puede ser por mínimos cuadrados.

Un método estudiado en [Paj01], para resolver el problema del movimiento basándose en puntos de interés, es el propuesto por Thompson. El proceso de búsqueda de correspondencia es iterativo y comienza con la detección de todos los posibles pares de correspondencias en imágenes consecutivas. Thompson en [TB81] impone un límite a la velocidad, lo que reduce el número de posibles correspondencias. Cada par bajo correspondencia tiene asignado un número, que representa la probabilidad

de su correspondencia. Este proceso se repite de forma iterativa hasta obtener un conjunto óptimo de correspondencias. El proceso termina si cada punto de interés en una imagen previa se corresponde precisamente con un punto de interés en la siguiente imagen y además, si la probabilidad global de correspondencia entre pares de puntos es significativamente más alta que otras posibles correspondencias o si la probabilidad global de correspondencia es mayor que un umbral prefijado o si la probabilidad global de correspondencias proporciona un máximo óptimo de probabilidad de todas las posibles correspondencias.

Este proceso, en [Paj01], se resume en siete pasos:

1. Determinar los conjuntos de puntos de interés A_1 y A_2 de las imágenes I_1 e I_2 y detectar todas las posibles correspondencias entre pares de puntos $x_m \in A_1$ e $y_n \in A_2$.
2. Almacenar la información sobre todas las posibles correspondencias P_{mn} de los puntos x_m e y_n.
3. Inicializar las probabilidades P^0_{mn} de correspondencia basándose en la similitud local (si dos puntos se corresponden entonces sus vecinos también deberían corresponder).
4. De forma iterativa se determina la probabilidad de correspondencia de un punto x_m con todos los posibles puntos y_n. Esto se hace como una suma promediada de probabilidades de correspondencia de todos los pares consistentes x_k, y_l, donde x_k son los vecinos de x_m y la consistencia de x_k, y_l se evalúa de acuerdo a x_m, y_n.
5. Se actualizan las probabilidades de correspondencia para cada dos puntos x_m e y_n.
6. Se repiten los pasos (4) y (5) hasta encontrar la mejor correspondencia x_m, y_n para todos los puntos $x_m \in A_1$.
7. Los vectores de correspondencia c_{ij} forman el campo de velocidad del movimiento analizado.

En otro trabajo, presentado en [Día96], se desarrolla un método de estimación de movimiento sobre una secuencia de imágenes mediante la detección y encaje de puntos relevantes. El método implantado utiliza la intersección de líneas, esquinas y centros de aspectos circularmente simétricos, como características. El algoritmo se basa en el análisis estadístico de las orientaciones del gradiente sobre un entorno circular de la características o puntos relevantes. Una vez obtenidos los puntos relevantes

procede a distinguir agrupamiento de puntos y establecer la correspondencia entre los grupos detectados. Es decir se consideran los dos problemas, el de agrupamiento y el de correspondencia, de manera conjunta para dar una solución simultanea.

Existen trabajos más recientes como [PAE04], que utilizan la extracción de características con la finalidad de calcular el tiempo al impacto utilizando imágenes log-polares. Muchos otros trabajos siguen utilizando esta técnica para el análisis del movimiento [BN07], [DYMS06].

2.3.2. Métodos basados en la correspondencia de regiones

En general los métodos basados en la correspondencia o correlación de regiones son menos sensibles al ruido al tomar más datos de la imagen en el proceso de estimación del movimiento. Esta técnica es ampliamente utilizada y surgen nuevos trabajos de investigación que utilizan esta técnica [DYMS06]. Básicamente, esta técnica de análisis, consiste en minimizar la diferencia en el desplazamiento entre imágenes en un bloque de píxeles. El principio básico de este método consiste en dividir la imagen en una serie de regiones de igual tamaño. Para cada región se busca en la siguiente imagen su posible correlación, minimizando un criterio de error.

Es el método más utilizado para la compensación del movimiento en la mayoría de los estándares de compresión de video [KMK06]. Dentro de este tipo de métodos cae un algoritmo llamado *Algoritmo de Correspondencia por Bloques* (BMA por sus siglas en inglés de Block Matching Algorithm). Este algoritmo ha sido ampliamente estudiado en [BK00], y su uso se debe a su fácil implementación en hardware.

Además de su amplia utilización para la compensación del movimiento, en la mayoría de los estándares de compresión de video, su uso también es ampliamente aplicado en filtrado y restauración de imágenes.

En [Gha03] se explica el algoritmo BMA con la finalidad de realizar una reducción de datos a procesar basada en la redundancia temporal. En dicha bibliografía también se mencionan varias funciones de error como: la función de correlación por cruce (CCF Cross Correlation Function), la función de error cuadrático medio (MSE Mean-Square Error) y la función del error absoluto medio (MAE Mean Absolute Error).

En la figura 2.8 se muestra el problema de la estimación del movimiento como se presenta en los estándares de compresión de video. Dada una imagen de referencia y un bloque de $N \times M$ de la imagen actual, el objetivo para la estimación del

movimiento es determinar el bloque $N \times M$ en la imagen de referencia que tenga la mejor correspondencia (de acuerdo a un criterio dado, ecuación 2.33) con un bloque de la imagen actual. La imagen actual es la I_2 obtenida en un tiempo t. La imagen de referencia es la I_1 obtenida en un tiempo anterior $t - n$.

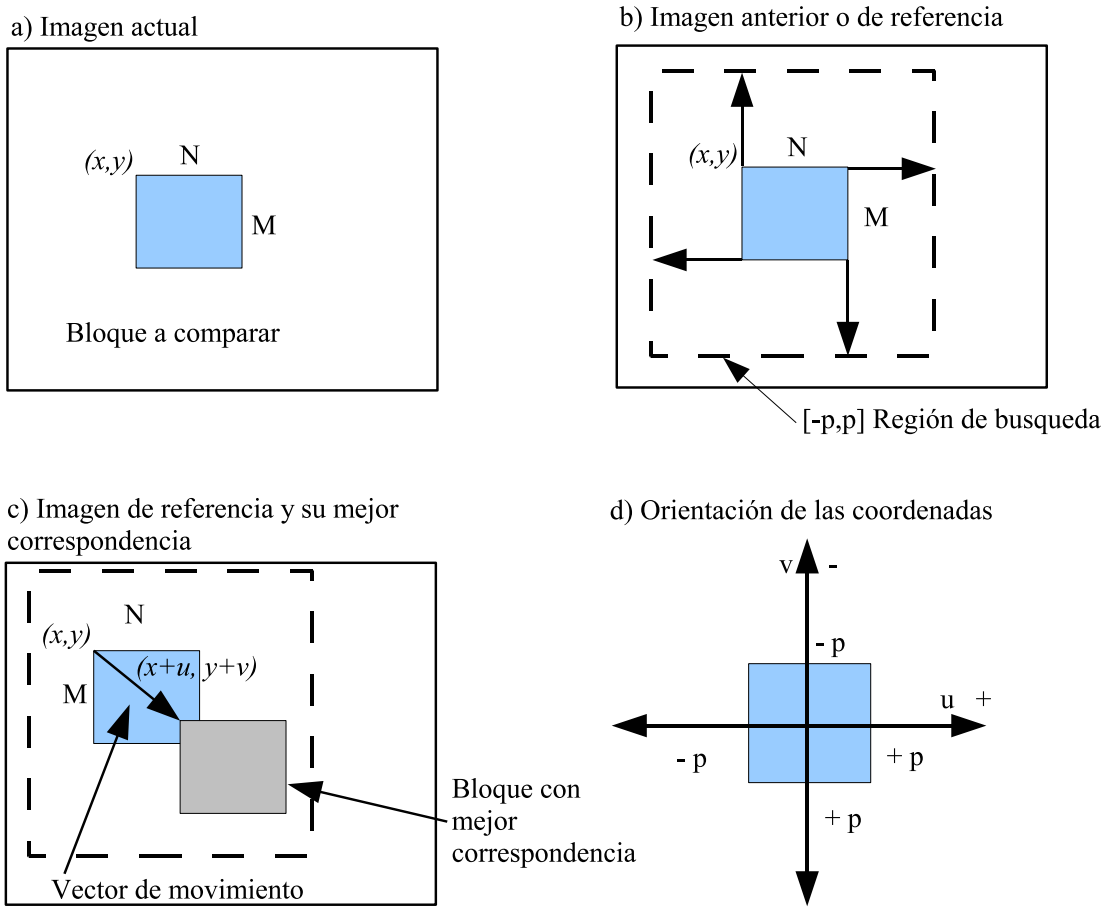


Figura 2.8: *Proceso de la estimación del movimiento mediante el algoritmo de correspondencia por bloques.*

La localización del bloque está dada por las coordenadas (x, y) partiendo de la esquina superior izquierda de la imagen. Existen algoritmos que realizan una búsqueda y correlación en toda la imagen (FSA por Full-Search Algorithm), pero dado que representa un elevado costo computacional se hace impráctico este algoritmo. Por tal razón el área de búsqueda se restringe a una región $[-p, p]$ (figura 2.8b) alrededor de la localización del bloque original de la imagen actual.

Sea $(x + u, y + v)$ la localización del bloque con la mejor correspondencia de la imagen de referencia (figura 2.8c). Además comúnmente el vector de movimiento

es expresado en coordenadas relativas, de tal manera que (x, y) se considera en la localización $(0, 0)$, entonces el vector de movimiento es simplemente (u, v). Esta consideración, de un desplazamiento común (u, v) para todos los píxeles dentro del bloque implica una restricción de suavizado local en el campo de movimiento. La restricción de suavizado local satisface sólo a bloques de tamaño pequeño. Por otro lado la selección de un bloque de dimensión pequeña de $N \times M$ ayuda a disminuir el tiempo de búsqueda de los vectores de movimiento, además de que pocos píxeles participan en el proceso de correlación.

Al trabajar con un tamaño de bloque fijo se limita el campo de movimiento estimado, ya que no hay posibilidad de operar bloques que contengan varios movimientos. Para subsanar esta deficiencia existen trabajos que utilizan la correlación de bloques en entornos de multirresolución.

Por otro lado existen nuevos algoritmos que buscan reducir al mínimo la complexidad computacional del algoritmo BMA. Estos nuevos algoritmos, denominados *fast motion estimation*, pretenden encontrar el bloque con mayor similitud más rápidamente, aun a riesgo de que la solución sea sólo un óptimo local [Gha03] [GG02] [YZKJ05].

2.4. La aproximación diferencial

Esta técnica es de las primeras en aparecer por su bajo costo computacional [YA81]. El principio básico de esta técnica es que considera que cualquier movimiento perceptible en la escena se traduce en cambios en la secuencia de imágenes tomadas de dicha escena, por lo cual, si tales cambios son detectados, se pueden analizar las características de este movimiento.

Estas técnicas son ideales para aplicarse en casos donde sólo se pretenda, por ejemplo, detectar la existencia de movimiento o sus componentes, si éste se produce en un plano conocido. Sin embargo también es posible utilizarla en combinación con otras técnicas como sustracción del fondo [Pic04], en el dominio comprimido [TcAA04], diferencias acumuladas y otras técnicas [HS07], [LLLL07], [MMN06], [AMMS06]. Esto es, debido a que su simplicidad para detectar la zona de interés permite concentrar un posterior esfuerzo computacional en el área detectada. Una revisión de algoritmos de detección por cambios en la imagen se pueden encontrar en [RAAKB05]. En ese trabajo el objetivo fundamental es identificar un conjunto de píxeles que han cambiado significativamente considerando una imagen de refer-

encia y la imagen actual. Adicionalmente se mencionan métodos de pre-procesado, modelos predictivos y modelado del fondo de la imagen entre otros temas.

La imagen diferencia I_d, se puede definir como:

$$I_d(\mathbf{p}, t_1, t_2) = I_2(\mathbf{p}, t_2) - I_1(\mathbf{p}, t_1) \tag{2.35}$$

donde $\mathbf{p} = (x, y)$ es un píxel de la imagen y t_1, t_2 son los instantes de tiempo de dos imágenes consecutivas. Nótese que los valores de la imagen diferencia pueden ser negativos. Una variante de la ecuación (2.35) consiste en formar la imagen resultante F_{out} mediante:

$$I_{out} = \begin{cases} I(\mathbf{p}, t_2) & \text{si } I_d(\mathbf{p}, t_1, t_2) \geq T_d \\ 0 & \text{en otro caso} \end{cases} \tag{2.36}$$

donde T_d es un umbral de detección de cambio. Esta variante permite retener sólo regiones de cambio significativo.

Las ventajas a destacar en este método de detección del movimiento son:

1. Se puede considerar como una aproximación a la derivada en el tiempo.
2. Se puede considerar un operador de localización de bordes. Esto sólo cuando la imagen diferencia se obtiene a partir de dos imágenes ligeramente desplazadas.
3. Segmentando la imagen diferencial puede estimarse la dirección del movimiento del objeto.
4. La extremada simplicidad de procesado.
5. Se presta a una cómoda implantación paralela.
6. Es muy adaptable a medios dinámicos.

Sus inconvenientes son: la información que proporciona no es demasiado descriptiva sobre la forma y movimiento de los objetos, y no es robusto frente a cambios de intensidad.

Sin embargo, este mecanismo para obtener la imagen diferencia, sí que permite detectar la zona de la imagen donde se están produciendo cambios, pudiéndose así concentrar el posterior esfuerzo computacional en el área detectada. Así, mediante el uso de una imagen de diferencias acumuladas, es posible recopilar la historia de movimientos a lo largo de un determinado intervalo de tiempo, pudiendo describir el movimiento. Otra alternativa, comúnmente utilizada en vigilancia, es que

mediante una cámara estacionaria se realizaría una substracción del fondo y todo aquello diferente al fondo se registraría como un movimiento.

2.4.1. Imágenes de diferencias acumuladas

La idea básica de realizar una acumulación de imágenes diferenciales es tener la historia de los movimientos durante un determinado periodo de tiempo. Este método de detección del movimiento es uno de los primeros en implementarse [JN79] por su bajo costo computacional. Jain y Nagel proponen calcular la imagen diferencial I_T mediante:

$$I_T(\mathbf{p}, t_n) = I_d(\mathbf{p}, t_{n-1}, t_n) - I_T(\mathbf{p}, t_{n-1}) \quad para \quad n \geq 3 \tag{2.37}$$

donde

$$I_T(\mathbf{p}, t_2) = I_d(\mathbf{p}, t_1, t_2) \tag{2.38}$$

Este proceso recursivo permite acumular en I_T la información del movimiento en el periodo del tiempo seleccionado. El mismo autor en [Jai84] presenta un análisis de varias alternativas para el caso donde tanto la cámara como los objets se encuentren en movimiento. En dicho trabajo se discuten los métodos para combinar la información obtenida de la diferencia positiva, negativa y absoluta y los cuadros acumulativos de las imágenes diferenciales. También presentan un acercamiento para la segmentación de escenas dinámicas usando información disponible en la diferencia y las imágenes diferenciales acumuladas. Destacan la eficacia del método y proponen la posibilidad de implementarla en el tiempo real usando un hardware especial.

Cabe destacar que esta estrategia aun es utilizada pero en conjunto con alguna medida de análisis espacial. Sangi et al., en [SHS04], proponen utilizar alguna función de error aunado a un umbral en el proceso de la obtención de imagen diferencial.

2.4.2. Sustracción del fondo

Este método es considerado como una combinación de dos técnicas íntimamente ligadas. Una que se encarga de obtener el fondo y otra que se encarga de localizar la diferencia entre la imagen de referencia (fondo) y la imagen actual [CLK$^+$00].

Esta técnica es comúnmente utilizada en el área de vigilancia mediante cámara estacionaria que realizar una substracción del fondo de la escena. Todo aquello diferente al fondo de la escena se registraría como un movimiento y sólo cuando se registra un movimiento es cuando se realiza la captura de las imágenes para ser archivadas. También es posible encontrar esta técnica utilizando cámaras en movimiento adicionando algún mecanismo extra para solventar la extrema sensibilidad, en ambientes dinámicos, que presentan los algoritmos de substracción del fondo.

Collins et al. presentan un extenso trabajo llamado VSAM (Video Surveillance And Monitoring)[CLK+00]. En este trabajo presentan multiples algoritmo de substracción de fondo que considera 3 imágenes diferenciales. También desarrollan algoritmos tanto para el registro de movimiento en escenas con cámaras estáticas como para el seguimiento de objetos, con cámaras en movimiento.

En [GED04] implementan un algoritmo que detecta y sigue objetos en movimiento, sustituyendo la substracción del fondo por el histograma espacio temporal de los píxeles de un conjunto de imágenes diferenciales. El histograma es normalizado y calculan la entropía para encontrar el movimiento del objeto. Nuevos trabajos siguen apareciendo, utilizando esta técnica, y son utilizados en distintas aplicaciones [OD07], [FII+06] y [YCH07].

Capítulo 3

Arquitecturas de procesamiento de imágenes

3.1. Introducción

El procesamiento digital de imágenes ha sido un área de la visión artificial ampliamente estudiada y utilizada por diversas disciplinas tales como: medicina [DMC+07], biología [GSS07], astrofísica [LT05] e ingeniería [LWEG07]. La finalidad de realizar un procesado digital de las imágenes es el obtener información objetiva de la escena captada por un sensor óptico. En esencia, dos de las tareas fundamentales en el procesamiento digital de imágenes son: la mejora de una imagen digital, ya sea con fines interpretativos (reconocimiento) o de calidad de imagen (restauración, descompresión, etc.) y la toma de decisiones de manera automática, en función de la información que nos proporciona la imagen o las imágenes.

Las arquitecturas de procesamiento de imágenes, de forma general, constan de tres etapas bien definidas: etapa de adquisición, etapa de procesado y una etapa final, que transmite el resultado y ejecuta la acción para la cual el sistema fue diseñado. Las tres etapas se muestran de manera general en la figura 3.1.

La **etapa de adquisición**, que regularmente es un sensor óptico, se encarga de transformar la información visual externa en una señal que sea comprensible para un sistema digital. De esta manera, un sistema de adquisición consta de un sensor visual, una etapa de digitalización de la señal del sensor (en caso de ser necesaria) y un dispositivo de almacenamiento de la señal a procesar, comúnmente llamado *frame-grabber*. Es importante destacar que existen sensores "*avanzados*"

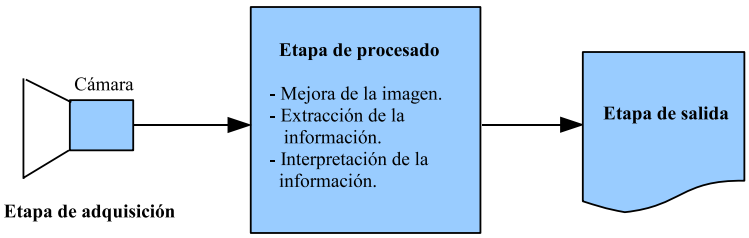


Figura 3.1: *Etapas que constituyen un sistema para el procesamiento de imágenes.*

que incluyen etapas de pre-procesamiento e incluso algún sensor puede realizar un procesamiento más elaborado o complejo. Dispositivos de este tipo se abordarán más adelante.

La **etapa del procesado**: según la función y/o robustez del sistema esta etapa puede constar de varios mecanismos para su procesado. Estos mecanismos tienen diferentes objetos como es la de realizar una mejora de la imagen con el objetivo de producir representaciones más útiles en otros mecanismos del procesamiento. Por ejemplo, es posible realizar un filtrado a la imagen con el objetivo inmediato de realizar una extracción de características más eficiente de la imagen y finalmente conocer acerca del dominio particular en el que se está trabajando y poder identificar e interpretar la escena observada. Una vez que se haya interpreto la escena es posible organizar y controlar el flujo de la información hacia la última etapa del sistema de visión.

En la **etapa final** o de salida se visualizará el resultado o bien se utilizará la información para la toma de la decisión de manera automática. Todo ello en función de la información que proporcionó la o las imágenes adquiridas previamente, en la etapa de adquisición.

En este capítulo se presentarán, de manera general, las etapas de adquisición y procesado. Además se abordarán las arquitecturas implementadas por software y las arquitecturas implementadas por hardware, para el procesado de imágenes. Finalmente se concluirá el capítulo con arquitecturas específicas para el cálculo de flujo óptico.

3.1.1. Etapa de adquisición

El objetivo de la etapa de adquisición consiste en trasformar la energía de luz incidente en energía eléctrica o una señal que sea comprensible para un sistema

digital. De esta manera la etapa de adquisición consta de un sensor visual y una etapa de digitalización de la señal del sensor (si es necesaria).

Existen básicamente dos tipos de sensores utilizados [Dal05]. Los dispositivos de carga acoplada (*Charge-Coupled Devices* o CCD) y los sensores basados en la tecnología Metal Oxido Semiconductor Complementado (*Complementary Metal Oxide Semiconductor* o CMOS). A pesar de que los sensores basados en tecnología MOS y los CCD aparecieron casi simultáneamente a finales de los años 60's [Wan01], los sensores CCD son la tecnología más estudiada y que por la misma razón explotan sus cualidades al 100 por ciento.

La razón por la cual se prefirió impulsar el diseño de sensores basados en tecnología CCD, respecto a la tecnología MOS, fue porque la tecnología CCD presentaba una mayor escala de integración. El poseer una mayor escala de integración se reflejaba en una mayor calidad de la imagen adquirida. Ese hecho hizo que los sensores CCD fueran considerados como la alternativa más factible, en ese momento, convirtiéndose en los dispositivos dominantes, y que su estudio y desarrollo se encuentre actualmente en una fase tecnológica madura.

De hecho los dispositivos CCD y CMOS utilizan los mismos mecanismos para la captura de la luz, el mismo principio fotoeléctrico. Ambos utilizan silicio como elemento de diseño y un conjunto de fotosensores que representan la intensidad de la señal óptica. La diferencia entre estos dispositivos radica en cómo y cuándo se realiza el proceso de obtención de su carga a su equivalente voltaje/corriente. Este hecho es atribuible a su proceso de fabricación. Mientras que la fabricación de un sensor basado en tecnología CMOS es un proceso de fabricación estándar, para un sensor CCD requiere un proceso de fabricación especial, que suele ser más costoso. Adicionalmente éstos requieren generalmente otro chip para proporcionar una información de salida digital.

La tendencia actual en la utilización de sensores está cambiando paulatinamente y conforme avanza el desarrollo tecnológico se empiezan a explotar mejor las cualidades de los sensores basados en tecnología CMOS. No obstante hoy en día la ventaja más importante de los sensores CCD es la excelente calidad de las imágenes obtenidas. Aunque el avance tecnológico hace que cada vez más dicha ventaja se estreche. Aunado a esto las características propias de la tecnología CMOS está logrando que los sensores basados en dicha tecnología sean más utilizados. De esta manera los sensores CMOS gradualmente se están posicionando como los sensores de la nueva generación e incluso se ha pensado que los sensores CCD han llegado a

CCD	**CMOS**
Tecnología madura	Tecnología reciente
Excelente calidad de imagen	Buena calidad de imagen
Excelente inmunidad al ruido	Buena inmunidad al ruido
Acceso secuencial	Acceso aleatorio a píxel
Alto consumo de potencia (relativo)	Bajo consumo de potencia
Alto coste de producción	Bajo coste de producción
Baja escala de integración	Alta escala de integración

Cuadro 3.1: *Características comparativas entre sensores CCD y CMOS.*

su fin [Bla01].

De manera general, se pueden resumir las características de los sensores CCD y CMOS en el cuadro 3.1. En dicho cuadro se puede observar que el sensor CCD presenta una mejor calidad de imagen. Por tal razón, es importante destacar que la *calidad* de las imágenes es relativa al tipo de aplicación que se le piense dar al sistema de procesamiento de imágenes. No es lo mismo un sistema de visión para aplicaciones microbiológicas o de microscopía óptica, que un sistema de visión para aplicaciones de detección de movimiento en sistemas de seguridad o de navegación de vehículos autónomos. En las primeras aplicaciones se requiere una alta calidad y extremada sensibilidad en las imágenes capturadas. En las últimas aplicaciones los objetos a visualizar presentan características tales que no demandan la necesidad de utilizar un excelente sensor, pues basta con un buen sensor.

A pesar de lo antes dicho existen áreas específicas de la microscopia óptica que presentan una tendencia a utilizar cada día más con imágenes digitales y utilizan sensores CMOS[1]. La razón es, como se dijo anteriormente, que las calidades de las imágenes obtenidas mediante sensores CCD y CMOS día a día se estrechan más. En [PGP+00] se realiza un estudio donde se muestra que las imágenes capturadas con sensores CCD y con sensores CMOS-APS (*CMOS-Active Píxel Sensor*) presentan una calidad similar. Dicho estudio fue hecho para usos en técnicas exploratorias de radiología periapical (radiología intrabucal). En [KSF03] también se muestra un estudio comparativo en técnicas que utilizan imágenes obtenidas con sensores CCD y CMOS. En dicho estudio concluyen que respecto a la calidad de imagen ambos tipos de sensores producen imágenes radiográficas con similar calidad para su aplicación.

Se puede decir que existen aplicaciones donde los requisitos necesarios del sensor

[1]http://micro.magnet.fsu.edu/primer/digitalimaging/index.html

óptico son considerados como satisfactorios o de buena calidad siempre y cuando las imágenes capturadas satisfagan las necesidades de una función específica del sistema. Dicho esto, la calidad de la imagen adquirida para el sistema es "*relativa*" a la aplicación, y no requiere en muchas ocasiones un excelente sensor para la captura de las imágenes.

Para este trabajo, la tecnología CMOS ofrece una buena *calidad* de imagen. Por esta razón se opta por escoger una cámara digital. En particular se utiliza la cámara Dragonfly Express, de Point Grey Research[2]. Esta cámara captura más de 200 frames por segundo (fps), en formato 640×480, con 256 tonos de gris (8 bits) y tiene la opción de acceder a una área específica de la imagen.

Sin embargo, es importante resaltar que las ventajas de la tecnología CMOS la hacen muy competente para la próxima generación de sensores, sobre todo para aplicaciones en la robótica móvil. Dentro de las ventajas, de los sensores CMOS, existen tres características muy importantes: bajo consumo de potencia, bajo coste de diseño y alta escala de integración. El bajo consumo de potencia y sobre todo el bajo coste estimula el diseño, desarrollo y la variedad de aplicaciones tanto para sistemas móviles como para nuevas aplicaciones. Pero la característica más sobresaliente es su alta escala de integración, permitiendo incorporar otros componentes en el mismo chip pudiendo desarrollar los llamados *smart sensors*. Dicho de otro modo se pueden incorporar dentro del mismo circuito integrado etapas que realicen un pre-procesado de la imagen, como filtrado, segmentación e incluso existen algunos sensores que integran algún tipo de procesado más complejo como es la extracción de características, formas, etc.

Finalmente otra ventaja agregada, que sin lugar a duda muchas veces es de gran utilidad, es que los sensores CMOS permiten un acceso aleatorio a los píxeles. Eso ha permitido explorar otras alternativas de los sistemas de visión al acceder y procesar solo unos píxeles de toda una imagen o bien obtener mayor resolución en zonas específicas o de interés de la imagen y en otras zonas tan solo realizar un submuestreo.

Smart sensors

Los dispositivos llamados *smart sensors* son circuitos que integran los elementos fotorreceptores y algún tipo de procesamiento en el mismo dispositivo, estando

[2]http://www.ptgrey.com/

fuertemente ligados en todos los niveles de procesamiento. Existen varias formas de clasificar estas microarquitecturas. Una de éstas puede ser en función del paralelismo de la arquitectura o del nivel de interacción entre la captura y el procesado de la información y consiste en tres clases [Moi00]:

1. **Procesado por píxel.** También conocidos como *smart pixels*. En este tipo de arquitectura cada píxel posee un elemento de proceso. Es sin lugar a duda la arquitectura que presenta el procesamiento paralelo más eficientemente.
2. **Procesado por columna.** Aquí existirá un procesador por cada columna de píxeles.
3. **Procesado por toda la imagen.** En esta arquitectura existe un procesador para toda la matriz de píxeles de toda la imagen.

Por otro lado, también pueden clasificarse dependiendo del algoritmo que sea utilizado por el dispositivo. Básicamente existen dos clases de algoritmos:

1. **Procesamiento espacial.** En este tipo de algoritmos la salida de cada píxel en un tiempo dado depende de los valores de las entradas de sus píxeles vecinos.
2. **Procesamiento espacio-temporal.** En este caso la salida de cada píxel en un tiempo dado dependa de los valores de las entradas de sus píxeles vecinos con un intervalo de tiempo adicional.

Sin embargo, cualquier tipo de *smart sensor* tiene el mismo objetivo, mejorar globalmente la sensorización.

Un *smart vision chip* que lleva a cabo el procesado por píxel se muestra en [KNL+01]. En el trabajo se presenta la idea básica de una tecnología de integración tridimensional inspirado en estructuras y funciones biológicas. El trabajo básicamente es tecnológico detallando aspectos de la técnica utilizada en la arquitectura. Se deja abierto dicho trabajo para aplicaciones de propósito general, en sistemas que requieran un alto poder de cómputo para el procesamiento de las imágenes.

En [FS05], se presenta una arquitectura en la cual se procesa toda una matriz de píxeles (toda la imagen) previamente capturados. Se indica que la arquitectura es inspirada biológicamente y los píxeles se procesan de manera secuencial, llamándose en el trabajo (*Marching-Pixels*). En [YB05], se implementó un chip de visión con procesado espacial, basado en tecnología CMOS. El sensor consiste en una retina de silicio de 60×100 píxeles. La característica particular es que la distribución de los píxeles no es rectangular si no que presenta una distribución log-polar. El área

de visión estaría distribuida en 60 anillos de 100 píxeles cada uno. Esta distribución originaría una mayor resolución en el centro del sensor y una menor detalle en la periferia. Por otro lado en [Sto06] se realizó un sensor que ejecuta un procesado espacio-temporal, en particular se implementa la ecuación de flujo óptico. En el trabajo se diseño un sensor de 30×30 píxeles utilizando tecnología BiCMOS de $0{,}8\mu m$, doble metal, doble polisilicio. Se especifica que posee una frecuencia de muestreo de 67 fps. pero que podría alcanzar mayores frecuencia de muestreo-procesado algo que ninguna otra arquitectura hardware podría proporcionar hoy en día, para este tipo de algoritmo.

En otros trabajos como [OA03], se desarrolla un sensor con un amplio rango dinámico para el ajuste en la sensibilidad del sensor, con la finalidad de evitar un filtrado de la imagen o un pre-procesado de la imagen para ser utilizada en el sistema para un proceso posterior. Y en [BR05], se presenta una retina CMOS con capacidad de procesar en tiempo real las imágenes. El circuito integrado CMOS captura las imágenes y las procesa extrayendo las características y el realce de los bordes mediante convoluciones 1D de Gabor.

La ventaja indudable de estas arquitecturas es la alta velocidad de procesado y la posibilidad de diseñar a la medida el algoritmo a implementar, pero sin tener la posibilidad de la reconfiguración del dispositivo y con un elevado coste de fabricación.

3.1.2. Etapa de procesado

El objetivo de la visión artificial es conseguir que las computadoras sean capaces de ver. Se sabe que el cerebro humano construye representaciones internas de los objetos de su entorno a partir de las percepciones que recibe de éstos, como contornos, color, textura, profundidad, movimiento, etc. Así mismo los sistemas de visión artificial intentan llevar a cabo una imitación de dicho proceso, clasificando la información obtenida en tres niveles [Luc03]:

1. **Representación de bajo nivel:** A este nivel se asocian todos los datos de entrada, que pueden ser obtenidos mediante cualquier sensor óptico. Una vez que se capturan estas imágenes *puras* se les aplica algún proceso con objeto de producir información más útil para su futuro procesamiento.
2. **Representación de nivel medio:** A este nivel se asocian las imágenes segmentadas, obtenidas a partir de la representación de bajo nivel, mediante la agrupación de características: contornos, texturas o formas. Su obtención es

importante para tener noción del dominio particular en el que se esté trabajando.

3. **Representación de alto nivel:** En este último nivel el procesamiento evoluciona de la representación a la identificación y al análisis de la escena observada. El flujo de control hacia niveles inferiores desempeña en este punto un papel importante de cara a la realimentación del sistema.

Desde los años 70's y hasta los 80's el interés de la visión artificial se centraba en construir sistemas de visión de propósito general [Luc03] [Cob01]. Se desarrolló toda una serie de algoritmos de análisis, que permitían extraer las características de bajo nivel de las imágenes. Posteriormente se empezó a estudiar la forma de organizar la información y agrupar las características mas significativas de la imagen, como esquinas, fronteras o fragmentos de superficie, con la finalidad de identificar los objetos y deducir sus propiedades.

Todas aquellas investigaciones dejaron ver que la construcción de sistemas completos de visión eran una meta demasiado ambiciosa en aquel momento. Lo que hizo que se modificara la estrategia de desarrollo, centrándose en dominios más restringidos.

En la década de los 90's, gracias en parte al avance tecnológico, se desarrollaron diferentes modelos de visión artificial que permitían agrupar las características de nivel medio/bajo para obtener agrupaciones de nivel medio/alto. De esta manera, se inicia una área de investigación donde ya se podía tratar a un sistema de visión como un proceso completo, aunque con ciertas restricciones, sobre todo en los tiempos de ejecución (procesado en tiempo real).

3.2. Clasificación de las arquitecturas para procesado

Existen distintas formas de clasificar las arquitecturas para el procesamiento de imágenes. Una de ellas podría ser en función del nivel de complejidad de los algoritmos de procesamiento como lo proponen en [Bol00] y [KACB05]. Una segunda forma sería en función de cómo se procesan los datos y de cómo se procesan las funciones como lo propone Sima en [SFK97], para la clasificación de los sistemas de visión por computador. Otra forma de clasificar las arquitecturas, para el procesamiento de imágenes, podría ser en función de la capacidad de procesamiento paralelo de

la arquitectura utilizada, como lo hace Flynn en la taxonomía de arquitecturas de computadores [Fly95].

Dada la característica de este trabajo y las herramientas a utilizar para llevar a cabo una arquitectura eficiente de procesamiento, la clasificación propuesta es en función de la técnica utilizada en la implementación del algoritmo de procesamiento de imágenes. De esta manera la taxonomía propuesta en este trabajo consta de tres clases:

1. **Implementación software secuencial.** Es aquella que se implementa en un computador de propósito general (*monoprocesador*). Se puede decir que la arquitectura clásica utilizada para el procesamiento de imágenes. Dentro de la taxonomía propuesta por Flynn sería del tipo SISD. Aquí el procesamiento de la información es puramente secuencial.
2. **Implementación software paralela.** Esta clase basa su arquitectura en una plataforma software. A diferencia de la clase anterior, esta clase está conformada por un conjunto de varios computadores o procesadores interconectados entre si. Esta clase se pueden enmarcar dentro de los sistemas paralelos SIMD o MIMD, propuestos en la taxonomía de Flynn, tales como los multiprocesadores, multicomputadores. También existen otras arquitecturas paralelas como procesadores vectoriales, matrices sistólicas, entre otras.
3. **Implementación hardware.** Este tipo de clase puede tomar cualquier arquitectura existente, para el procesado de imágenes, según la necesidad del algoritmo o del sistema. La razón es debido al hecho de que son sistemas dedicados para un propósito específico. Pueden utilizar dispositivos diseñados a la medida (diseño *full-custom*), pero también dispositivos programables como los ASIC's o FPGA's (diseños *semi-custom*).

Dado el avance tecnológico la distinción entre cada clase cada vez es más difícil de distinguir. Incluso se podría considerar una nueva clase que podría ser de *implementación híbrida*. Esta nueva clase sería una arquitectura que puede combinar dos clases diferentes. Un claro ejemplo, es el uso de las FPGA's en conjunto con otras arquitecturas de propósito general.

3.2.1. Implementación software secuencial: la aproximación clásica

Actualmente se podría considerar como una arquitectura clásica, para el procesamiento de imágenes, a un sistema que utilice como unidad principal para el procesamiento a un computador de propósito general con una tarjeta aceleradora gráfica.

Dentro de la clasificación de las arquitecturas de computadores, propuesta por Flynn en [Fly95], sería una arquitectura del tipo SISD. En estas arquitecturas un único procesador interpreta una única secuencia de instrucciones para operar con los datos almacenados en una única memoria. Se le denominó implementación software debido a que necesariamente requiere un sistema operativo y un programa que ejecute la secuencia de instrucciones para lograr un propósito específico, dentro de una arquitectura de propósito general o PC. Este tipo de arquitecturas es ampliamente utilizado para aplicaciones sencillas, aunque dado al avance tecnológico cada vez pueden desempeñar funciones más complejas.

Ejemplo de lo antes dicho se encuentra en los trabajos [YB03] y [YB05]. En el primer trabajo se desarrolla un sistema de visión activa monocular utilizando un sensor log-polar. El segundo trabajo consiste en una mejora del primer trabajo, creando un sistema más eficiente. El algoritmo empleado es la transformada log-Hough. El diseño es implementado en un computador PC Pentium III 600 MHz. Puede procesar hasta 20 imágenes por segundo debido a que utiliza un sensor log-polar para disminuir la cantidad de información a procesar, ya que utiliza imágenes de 60×100 píxeles, en coordenadas log-polares.

En otro trabajo se desarrolla un sistema de video vigilancia móvil [FMP05], que se implementa sobre una Laptop Athlon 2 Ghz, 256 Mbytes en RAM. Puede trabajar en tiempo real, procesando 25 imágenes por segundo de 320×240 píxeles. Para poder lograr estas frecuencias de procesado se deben utilizar algoritmos que requieran poco poder de cómputo. En el trabajo primero se calcula la imagen diferencial (diferencia entre dos imágenes consecutivas). Posteriormente se utilizada una red neuronal para clasificar la imagen resultante, ya sea entre una escena estática o una escena con objetos móviles. En el mismo trabajo se deja abierta la opción de aplicar otro proceso para el reconocimiento de personas lo que conllevaría a tener un coste más elevado en el procesado, lo que posiblemente imposibilitaría usar la misma arquitectura hardware de procesamiento.

La desventaja de este tipo de arquitecturas es su limitado ancho de banda para

procesar grandes cantidades de información, que muchas veces es el impedimento para realizar procesado en tiempo real. Esta situación es peor aun si se considera implementar algoritmos con un elevado coste computacional o sistemas más elaborados. Sin lugar a duda las tarjetas aceleradoras gráficas aumentan considerablemente la velocidad de procesado, comparado con computador monoprocesador de propósito general, pero puede no ser suficiente para procesar algoritmos complejos como los de flujo óptico o tareas medianamente complejas que requieran procesar información en tiempo real.

Comúnmente los sistemas para el procesamiento de imágenes se dividen en varias etapas y en cada etapa se producen resultados o imágenes parciales que requieren ser almacenadas. La gran cantidad de información que debe ser almacenada temporalmente y como la información debe fluir rápidamente, en cada etapa del proceso, se puede generar un cuello de botella en el sistema, por lo que la arquitectura de procesamiento clásica se ve desbordada por las necesidades computacionales de los sistemas de procesamiento. De esta manera, la demanda de mayores capacidades de procesamiento computacional y la de un procesamiento más eficiente de la información, ha promovido la búsqueda de arquitecturas alternativas para el procesamiento de imágenes.

Es importante destacar que además de la forma de implementación del sistema de procesamiento de imágenes también hay que considerar el cómo se procesa la información. Esto se refiere a la manera de cómo el algoritmo va a manejar los datos, que puede ser mediante una ejecución serie, paralela o por flujo de datos. Estos conceptos se abordaran con detalle en la sección 3.2.3.

3.2.2. Implementación software paralela

En este tipo de arquitecturas siempre existirá un sistema operativo y un programa que controle, administre y lleve a cabo la comunicación entre las diferentes unidades de procesamiento, aparentando una sola arquitectura hardware. Estas arquitecturas se pueden enmarcar en los sistemas paralelos, los cuales están representados por los computadores que poseen arquitecturas SIMD y MIMD, según Flynn en [Fly95].

En la arquitectura SIMD se puede ejecutar en un ciclo de reloj una única operación. En la arquitectura MIMD se puede ejecutar, en el mismo ciclo de reloj, el mismo número de operaciones que de elementos de proceso posea la arquitec-

tura. Las operaciones, que se ejecutan, pueden o no ser idénticas y los elementos de proceso pueden ser procesadores o computadores independientes.

De esta manera la arquitectura MIMD es la más representativa de las arquitecturas paralelas. Dentro de la misma arquitectura existen los sistemas que poseen múltiples procesadores y aquellos que se basan en múltiples computadores autónomos interconectados entre si, simulando una arquitectura única.

Los sistemas basados en *multiprocesadores* consisten básicamente en dos o más procesadores similares. Todos los procesadores comparten la misma memoria principal y los mismos dispositivos de E/S. Una característica importante es el tiempo de acceso a memoria, que es aproximadamente el mismo para cada uno de los procesadores. El sistema está controlado por un sistema operativo integrado, que proporciona la interacción entre los procesadores y sus programas. La ventaja de estos sistemas son: su fácil gestión y configuración, poco consumo de potencia, poco espacio físico, son plataformas estables y que su modelo es más parecido al de un computador clásico, monoprocesador. Su desventaja radica en el límite de procesadores que puede utilizar y que su coste es mayor a otras arquitecturas paralelas. Recientemente el avance tecnológico ha hecho aparecer nuevos dispositivos microprocesadores con doble núcleo, denominados *procesadores duales*. Éstos consisten en dos microprocesadores en un mismo chip y pueden caer dentro de los sistemas monoprocesador si es considerado como un único chip o dentro de los sistemas multiprocesador a pesar de estar en un único chip.

Los sistemas basados en *multicomputadores* consisten básicamente en un conjunto de computadores independientes uno de otro pero que se encuentran interconectados y trabajan juntos como un solo recurso de cómputo. Esto proporciona la ilusión de ser una única máquina que trabaja para un único propósito. Su unidad de interacción física es normalmente un mensaje o un fichero completo. La indudable ventaja de este tipo de arquitecturas radica en su escalabilidad e incrementabilidad, también en que son superiores en términos de disponibilidad. Dentro de sus desventajas las más importantes son: el espacio físico requerido y su alto consumo de potencia. Ciertamente sus desventajas los hacen menos deseables para aplicaciones en robótica móvil, pero su uso en otras áreas de la visión artificial, donde no importe el tener sistemas grandes o voluminosos, va creciendo más día a día.

Ejemplos de estas arquitecturas, tanto de *microprocesadores* como de *microcomputadoras* y otros tipos de estructuras SIMD o MIMD, son presentadas a continuación.

En [FM04] se propone una arquitectura paralela basada en múltiples unidades de procesamiento gráfico (GPU's). En ella se ejecutan tareas de reconocimiento de patrones y algoritmos de visión artificial, que muestra un procesamiento varias veces más rápido que si se utilizara un computador convencional. Además se destaca la característica que poseen los GPU's en referencia a la memoria que poseen, como una cualidad y se destaca como una ventaja respecto al limitado ancho de banda que posee un computador estándar.

En [SKG01] y en [SKG02] se presenta un software (librería) que permite desarrollar, de forma transparente para el usuario, aplicaciones para el procesamiento de las imágenes sobre sistemas paralelos. El tipo de arquitectura paralela para la cual se desarrollo dicho trabajo es una MIMD, de memoria distribuida.

Otro sistema que basa su arquitectura en una estructura MIMD, del tipo multicomputador, es el utilizado en el laboratorio LIRA-Lab[3]. En este caso se utilizan una arquitectura multicomputador que funciona sobre una plataforma software llamada Yet Another Robot Platform YARP[4] [MFN06]. El sistema consiste de un conjunto de computadores conectados entre si mediante una red Ethernet Gbit. En cada módulo se procesa un algoritmo distinto para llevar a cabo en conjunto el funcionamiento de un robot que posee características especiales especificadas en [MCF+06]. Entre los algoritmos que se procesan en los diversos módulos están los de procesamiento de imágenes, de control de sus articulaciones, de audio y otros.

En [Til99] se implementa un sistema para realizar la segmentación de imágenes con una alta calidad pero con un alto coste computacional. El sistema es implementado en dos tipos de arquitecturas MIMD con el fin de comparar las prestaciones de dichas arquitecturas. Una arquitecturas del tipo multiprocesador, con memoria compartida y la otra arquitectura del tipo multicomputador, con memoria distribuida. En el trabajo se comparan las prestaciones de las arquitecturas llamada HIVE respecto la Cray T3E. La arquitectura HIVE está compuesta por 66 computadores Pentium Pro PC con 2 procesadores cada uno. La arquitectura Cray T3E es un supercomputador con 512 procesadores. Los resultados que mostraron fue que el algoritmo se ejecutaba aproximadamente $1,5$ veces más rápido sobre la HIVE la cual coste cerca de 86 veces menos que la Cray T3E.

Existen trabajos de procesamiento de imágenes, como [BL00], en el cual se desarrolla un sistema paralelo para la restauración de imágenes que utiliza una ar-

[3]http://www.liralab.it/

[4]http://yarp0.sourceforge.net/

quitectura multiprocesador y simultáneamente una arquitectura multicomputador. La primera de ellas es una supercomputador Cray T3E con 128 procesadores, la segunda es un cluster de 6 estaciones de trabajo interconectadas mediante una red Ethernet. En el trabajo, se demostró el buen rendimiento del sistema con distintos tipos de imágenes.

Además de las arquitecturas antes mencionadas otras arquitecturas, que también pertenecen a esta clase, pueden ser sistemas que, de una u otra forma, basan su plataforma en un software como control del sistema y utilicen conjuntamente un hardware para que desempeñe una etapa o función específica del sistema de procesado.

En [BIMK04] se presenta una arquitectura basada en un módulo software y un módulo hardware. El módulo hardware es implementado en una FPGA Virtex-E V2000 en la cual se realiza la extracción de características, de las imágenes del sistema, en paralelo. El módulo software recupera las características obtenidas y realiza operaciones adicionales para el reconocimiento de patrones. Todo esto ayuda para realizar un proceso en tiempo real.

En [YB05] se presenta un sistema para el seguimiento de objetos, utilizando un sensor óptico log-polar. Así se evita el uso de un hardware adicional para realizar la transformación de la imagen rectangular capturada a su equivalente en coordenada log-polar. La información es recibida por un computador Pentuim III a 600 Mhz, pudiendo procesar 20 fps. La desventaja en este tipo de sistemas radica en el uso de un sensor específico log-polar con una resolución fija, en este caso de 60×100 píxeles. Lo cual pierde flexibilidad el sistema en caso de intentar mejorar la resolución o bien para llevar a los límites del sistema. Esto es respetar un compromiso entre velocidad de procesado y la precisión del sistema.

La finalidad de estos sistemas, al tener un módulo software y un módulo hardware, tiene como objetivo tener una alta flexibilidad proporcionada por el software y un eficiente rendimiento y procesado al utilizar un hardware dedicado o específico para el sistema.

3.2.3. Implementación hardware

La ventaja indudable de estas arquitecturas es la alta velocidad de procesado y su posibilidad de diseñar a la medida el algoritmo a implementar. Existen una gran variedad de alternativas y dispositivos para llevar a cabo la implementación de un

sistema bajo este tipo de implementación.

La primera de ellas consiste en la utilización de los circuitos integrados VLSI diseñados a la medida (*full-custom*). En la sección 3.1.1 se detallaron estos sensores, llamados *smart sensors*, que consisten en sistemas completos para el procesamiento de imágenes dentro del mismo circuito integrado. Entre algunos de éstos *smart sensors* están: [KNL+01] [FS05] [YB05] [OA03] [BR05] y [Sto06].

El gran problema de los sistemas diseñados mediante circuitos hechos a la medida es el alto coste de fabricación y la pérdida de flexibilidad para una posible reconfiguración, en caso de que el sistema requiera futuras modificaciones.

Un claro ejemplo de la pérdida de flexibilidad de los circuitos integrados hechos a la medida, se puede observar en los trabajos [SGPB04] y [YB05], donde se aprecia claramente este hecho. En el primer trabajo se utilizó como sistema de captura una cámara digital de 640×480 píxeles. Al adquirir las imágenes eran transformadas en tiempo real a su equivalente log-polar, mediante una arquitectura reprogramable FPGA, teniendo la flexibilidad de poder obtener imágenes log-polares con distintas resoluciones. Mientras que en el segundo trabajo, en [YB05], se utilizó directamente un sensor óptico log-polar (basado en tecnología CMOS) el cual sólo podía capturar imágenes con una resolución máxima de 60×100 píxeles. En el posible caso de que existiera la necesidad de contar con una mayor resolución, el sistema requeriría un nuevo sensor. Este hecho generaría la necesidad de un nuevo dispositivo lo que conllevaría a un aumento en el coste del sistema, tanto económico como de tiempo. Por tal razón, muchas veces, el uso de dispositivos hechos a la medida VLSI (*full-custom*) son imprácticos.

Sin embargo, y gracias al avance tecnológico, existen una gran diversidad de alternativas para el diseño de sistemas con costes más accesibles y con una mayor flexibilidad para el diseño. Estas plataformas flexibles, que ofrecen la eficiencia funcional del hardware y la programabilidad del software, maduran tán rápidamente como la capacidad lógica de los dispositivos programables sigue a la Ley de Moore y al avance tecnológico de las técnicas de diseño de circuitos integrados.

En [TB01], y más recientemente en [Ma03], se presenta un resumen del estado del arte de las diferentes plataformas para el procesado de señales e imágenes por medio de arquitecturas de computo reconfigurable. Los dispositivos que se evalúan son los ASIC's, los DSP's, los procesadores de propósito general y las FPGA's. En ambos trabajos se desarrolló una comparación de la flexibilidad y las prestaciones de cada una de las plataformas. De estos trabajos se puede concluir con una tabla

Dispositivo	**Flexibilidad**		**Prestaciones**		
	Programa.	Reconfigura.	Área	Consumo	Cómputo
ASIC	Bajo	Bajo	Bajo	Bajo	Alto
DSP	Medio	Medio	Alto	Alto	Medio
μprocesador	Alto	Alto	Alto	Alto	Bajo
FPGA	Medio	Alto	Bajo	Medio	Medio

Cuadro 3.2: *Comparación de distintas plataformas de implementación para un sistema de procesado de señal e imágenes [TB01] y [Ma03].*

comparativa entre la flexibilidad y las prestaciones de las diferentes plataformas, como se muestra en el cuadro 3.2.

Del cuadro se puede observar que existen debilidades y fortalezas en cada una de las plataformas. Por tanto la plataforma a elegir dependerá del tipo de sistema de procesado de imágenes que se desee diseñar. Esto es, de las necesidades y características particulares del sistema a desarrollar. También se puede ver que cuanto mayor poder de cómputo posea el dispositivo menor será su flexibilidad de programabilidad o reconfigurabilidad. Esto mismo se ve claramente reflejado en la figura 3.2, en el trabajo desarrollado por Tessier en [TB01].

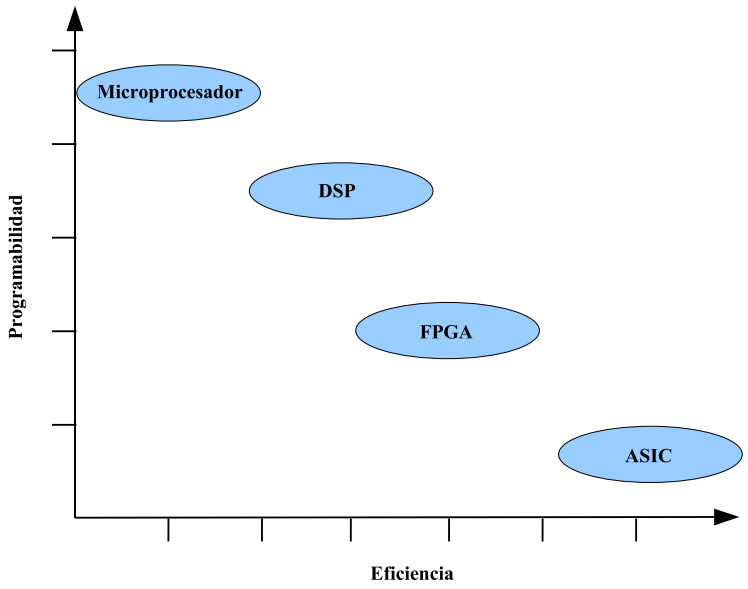


Figura 3.2: *Espectro de programabilidad - eficiencia del dispositivo, [TB01].*

Se puede apreciar que en un extremo se encuentran los microprocesadores, que

son dispositivos de propósito general, que presentan una mayor flexibilidad para programar al dispositivo. Pero son a su vez los que tienen la menor especialización, que permita llevar a cabo una ejecución eficiente del algoritmo. Esto indica que su capacidad computacional es limitada, en cuanto se refiere a un procesamiento masivo de información y procesado eficiente.

En el otro extremo se encuentran los dispositivos ASIC's. Estos dispositivos son lo más cercano a los dispositivos diseñados a la medida o VLSI. Son una alternativa utilizada para tareas de procesamiento de imágenes en tiempo real, que requieran una alta potencia de cómputo y que carezcan de necesidades de reconfigurabilidad. En [RV95] se describe la importancia de la utilización de los ASIC en tareas de procesamiento de imágenes. También trata otros aspectos como la metodología de diseño, pruebas y su cableado interno. Sin embargo, dada su poca flexibilidad de programabilidad y reconfigurabilidad hacen que sean poco utilizados en áreas de investigación, desarrollo y prueba de nuevos prototipos, de sistemas de visión artificial o de otra índole.

Así, se tiene que los DSP's y las FPGA's se encuentran en un punto intermedio, entre la flexibilidad de la reconfigurabilidad del software y la eficiencia funcional del hardware, que ofrecen los dispositivos hechos a la medida.

Las FPGA's son una alternativa que combina idealmente la flexibilidad del software y la velocidad del hardware, aproximándose cada vez más a la velocidad de un dispositivo ASIC. En muchos casos las FPGA's son utilizadas en combinación con DSP's. En [BMRA02] se presenta una arquitectura para el procesamiento de imágenes en tiempo real. La arquitectura paralela forma una matriz bidimensional de dispositivos reprogramables FPGA's y procesadores DSP's. Los procesadores son interconectados mediante una matriz sistólica 2D basado en unidades FPGA's.

En otro trabajo, en [MKN04], se desarrolló un sistema robusto que procesa en tiempo real un algoritmo para el seguimiento de objetos. La arquitectura está constituida por una FPGA y un DSP. Allí se indica que con pocos recursos hardware se pueden realizar procesos en tiempo real con una buena prestación y alta precisión.

Otro trabajo, donde realizan el diseño de una arquitectura exclusivamente con FPGA's se presenta en [THAE04]. Esa arquitectura alcanza una cantidad de operaciones, del orden de Giga operaciones por segundo. La arquitectura es una matriz sistólica 2D configurable para el procesamiento de imágenes en tiempo real. Así pues, siguen apareciendo nuevos trabajos que utilizan dispositivos FPGA's para implementar arquitecturas para el procesamiento de imágenes [DRM+06], [MJE07],

[DRP+06].

Con todo lo antes dicho es posible concluir que los dispositivos lógicos programables, como las FPGA's, son los que proporcionan el mejor equilibrio entre eficiencia y flexibilidad. Por esta razón se puede entender el hecho de que existan cada vez más trabajos y arquitecturas que utilizan este tipo de dispositivo, como es el caso de este trabajo.

Por otro lado es importante destacar que es un hecho sumamente importante el *considerar las características intrínsecas del algoritmo a implementar en el sistema.* Para dejar claro este principio es posible ejemplificarlo con dos casos: si el algoritmo a implementar es eminentemente paralelo, como podría ser el caso de una red neuronal artificial, y éste es ejecutado sobre una arquitectura secuencial tipo SISD (monoprocesador). Entonces se hace imposible ejecutar el algoritmo en paralelo. En este caso se estaría desaprovechando totalmente la característica intrínseca del algoritmo, por el tipo de arquitectura en que es implementado. El otro caso sería el opuesto. Si el algoritmo es eminentemente secuencial, como lo es una función iterativa, y es ejecutado sobre una arquitectura paralela, sea multiprocesador o multicomputador. Entonces la arquitectura no sería utilizada de manera óptima debido a la característica iterativa del algoritmo.

Por las razones anteriormente expuestas, se han propuesto nuevas topologías para acometer problemas orientados más al tipo de aplicación o a un tipo de algoritmo específico, donde el objetivo fundamental es diseñar arquitecturas de procesamiento que ejecuten algoritmos de la forma más eficiente como sea posible [SGFBP06a]. Esto ha hecho que las nuevas topologías propuestas requieran algún tipo de dispositivos programables, como los ASIC's o los PLD's, y/o circuitos integrados *full-custom.*

En [SKMD03] se muestran algunas alternativas o mecanismos universales de arquitecturas de procesamiento de datos paralelos, aplicados al procesado de imágenes. En él se proponen micro-arquitecturas con diferentes características basadas en una arquitectura reconfigurable. Entre las arquitecturas que trata están la Vectorial, la SIMD y la MIMD. En [BMRA02], también se muestra una clasificación más extensa de arquitecturas paralelas aplicadas a los sistemas de visión. Entre las clases que se destacan están las Sistólicas, las Vectoriales, las Asociativas, las SIMD's, las MIMD's (multiprocesador y multicomputador), entre otras.

Procesadores vectoriales y matrices sistólicas

Los procesadores vectoriales ejecutan de forma segmentada instrucciones sobre vectores y son comandados por un flujo de datos continuos. La computación vectorial surge debido a la necesidad de los computadores para resolver problemas matemáticos que requieren una precisión elevada y ejecutar programas que realicen de forma repetitiva operaciones aritméticas en coma flotante, con grandes matrices de números. Típicamente estos sistemas realizan cientos de millones de operaciones, en coma flotante, por segundo y cuestan entre 10 y 15 millones de dólares [Sta00]. Debido al elevado coste de dichos sistemas es poco práctico y habitual que existan diseños de sistemas de visión artificial, sobre todo para arquitecturas móviles o sistemas que requieran una autonomía de consumo de energía y una pequeña área para la implementación del sistema.

Por otro lado las matrices sistólicas consisten en un gran número de elementos de proceso (EP's) idénticos, con una limitada memoria local. Los EP's se colocan en arreglos lineales o de matriz y sólo se interconectan los EP's vecinos. En las matrices sistólicas los datos fluyen de un EP a su vecino o vecinos, según el tipo de arreglo, en cada ciclo de reloj. Durante ese ciclo de reloj cada EP realiza una operación sencilla.

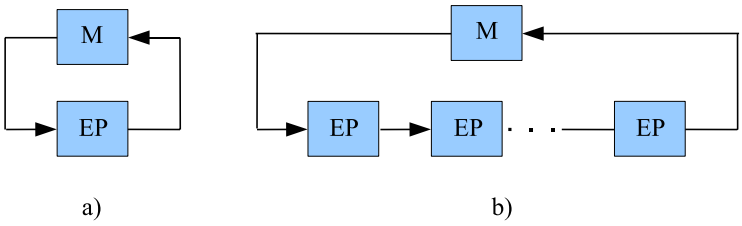


Figura 3.3: *Diagrama a bloques de: a)Arquitectura monoprocesador y b)Matriz sistólica lineal.*

Para apreciar mejor la diferencia entre una arquitectura convencional y una matriz sistólica, la figura 3.3 presenta ambas arquitecturas a bloques. La arquitectura monoprocesador 3.3a) y una matriz sistólica lineal 3.3b). Se puede ver que la matriz sistólica lineal reemplaza a un solo EP (procesador) por un conjunto de EP's (procesadores) de las mismas características. Dirigiendo así el flujo de datos para obtener un alto rendimiento con menos acceso a memoria.

Una arquitectura que utiliza este tipo de topología es implementado por Torres-Huitzil, et al. en [THAE04]. Allí se implementa una arquitectura sistólica sobre una FPGA, para el procesamiento de imágenes en tiempo real. La FPGA utilizada es un

dispositivo Virtex, XCV2000E la cual ejecuta aproximadamente 3,16 GOPS a una frecuencia de 60 MHz, permitiendo procesar hasta 120 imágenes por segundo (8,35 ms por imagen), de 512×512 utilizando mascaras de 7×7. El sistema desarrollado es reconfigurable y permite ejecutar varios algoritmos utilizados en el procesado de imágenes, como son: filtrado Gausiano, correspondencia de regiones y dilatación. En el mismo trabajo se presenta un cuadro comparativo entre su arquitectura, que utiliza una FPGA, y otras arquitecturas propuestas por distintos autores pero que utilizan otros tipo de dispositivos como son: los DSP's, ASIC's, multiprocesadores, procesadores vectoriales, procesadores superescalares, entre otros. Concluyen que la eficiencia de la arquitectura es similar y algunas veces superior que el resto de las arquitecturas propuestas. También se resalta que muchas de las arquitecturas utilizan un gran número de unidades de proceso y una cantidad considerable de memoria. Además se menciona que aunque el tiempo de procesamiento es un poco mayor que el propuesto por algunas arquitecturas que utilizan dispositivos ASIC's, destaca por mucho la ventaja de su flexibilidad de reconfiguración que poseen las FPGA's. Esta ventaja es con la finalidad de modificar ya sea el tamaño de ventana utilizada para el procesamiento, poder hacer variaciones a los algoritmos o implementar nuevos algoritmos.

Máquinas de flujo de datos

Hay dos formas de procesar la información, una es mediante la ejecución en serie de una lista de comandos y la otra es la ejecución de un comando o varios comandos demandados por los datos disponibles, llamada flujo de datos. El procesamiento de la información en serie tiene sus orígenes en la ejecución de las instrucciones utilizando un contador de programa, que es la base de la arquitectura propuesta por John von Neuman. La segunda forma de procesamiento de la información es más compleja en su diseño por el hecho de la necesidad de un circuito hardware de control más complejo. Y para casos de sistemas basadas en una plataforma software se requieren lenguajes de programación que soporten la programación concurrente, como son el Prolog, el ADA, etc. Para esta forma de procesado se ejecutan en el momento que se tienen los datos necesarios para ello, y se debería poder ejecutar todas las instrucciones que se demanden en el mismo tiempo.

En una arquitectura de flujo de datos una instrucción estará lista para su ejecución cuando los datos necesarios están disponibles. La disponibilidad de los datos se consigue por la canalización de los resultados de cada una de las instrucciones

ejecutadas, para todos los operandos de las nuevas instrucciones que esperan a ser procesados. Además, cada instrucción dentro de las arquitecturas de flujo de datos lleva los valores de las variables y la ejecución de una instrucción no afecta a otra que esté lista para ser procesada. De esta manera, varias instrucciones pueden ser ejecutadas simultáneamente permitiendo tener un alto grado de concurrencia y paralelización.

En la figura 3.4 se muestra un diagrama a bloques de una máquina de flujo de datos. Las instrucciones y los operandos se encuentran almacenados en una memoria de instrucciones y datos (MID). Cuando una instrucción está lista para ser ejecutada se envía a uno de los elementos de proceso (EP) a través de una red de arbitraje. Cada EP posee una pequeña memoria local y una vez que es procesada la instrucción se envía el resultado mediante una red de distribución.

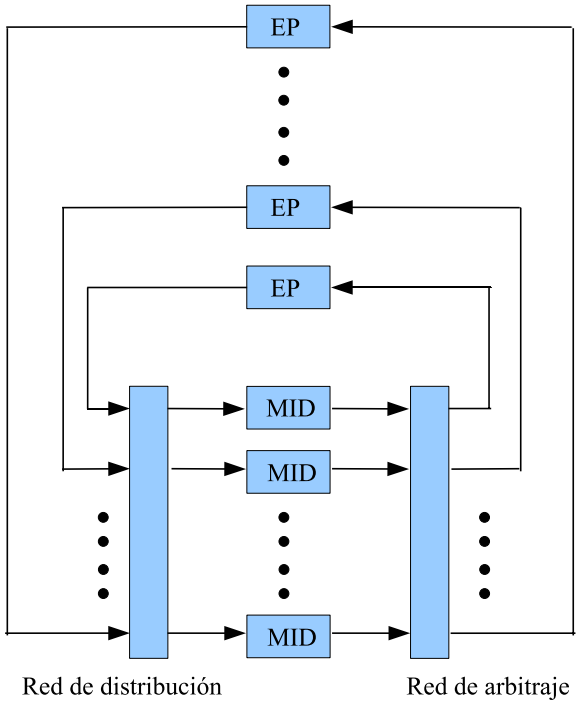


Figura 3.4: *Diagrama a bloques de una máquina de flujo de datos, donde EP es un elementos de proceso y MID es una memoria de instrucciones y datos.*

Se puede ver que las particularidades de las arquitecturas de flujo de datos son varias y están bien definidas. Por un lado no requiere tener una memoria compartida, un contador de programa o un control de secuencia del sistema. Por otro lado requiere un circuito que detecte la disponibilidad de los datos, un circuito que compruebe los datos y las instrucciones que necesitan esos datos y otro más que permita realizar,

de ser posible, una ejecución en cadena asíncrona de instrucciones.

Dada las características de este tipo de arquitecturas, muy alejadas de los sistemas de propósito general, existen muy pocas arquitecturas comerciales e incluso se podría decir que son sólo para aplicaciones de investigación. Uno de los primeros trabajos registrados en la literatura fue el VIM, propuesto en 1985 en el MIT [Guh85]. Este fue un sistema de cómputo experimental desarrollado en el MIT. Adicionalmente se desarrolló un lenguaje de programación para el mismo sistema denominado VIM VAL programs.

En 1991 se diseña un circuito procesador VLSI por flujo de datos, llamado DFP (*Data-Flow Processors*). Se diseña en tecnología CMOS, operando a 25 MBytes por segundo y alcanzando a procesar hasta 50 MOPS [QZ91]. Otro trabajo que utiliza este tipo de arquitectura es el desarrollado por [PLV96], y aunque no dice claramente que se trata de este tipo de arquitectura el autor le denomina arquitectura pipeline adaptable. Incluso se refiere a una representación no explícita de un flujo de control y un flujo de datos. Más recientemente, en [SK02], se presentó un modelo de computador para procesar imágenes en tiempo real basado en una metodología *síncrona* de datos.

Sin embargo, este tipo de arquitecturas no han sido muy estudiadas por su complejidad y existen muy pocas referencias. Por otro lado es evidente que son muy útiles para procesos que requieren un alto potencial de cómputo o que se desee realizar para algoritmos muy particulares. Esto hace que la utilización de dispositivos lógicos programables, para el diseño de estas arquitecturas, sean muy útiles dada su flexibilidad. Permitiendo proponer nuevas topologías con la finalidad de contar con sistemas altamente eficientes.

3.3. Arquitecturas para el cálculo del flujo óptico

Durante muchos años el cálculo del flujo óptico se ha llevado a cabo por computadores de propósito general e incluso, algunas veces, por computadores paralelos, sin haber logrado realizar el proceso en tiempo real. Todo eso debido al alto coste computacional que requiere el cálculo del algoritmo además de que muchas veces se busca que el algoritmo proporcione resultados con alta exactitud. Estas razones han hecho que innumerables técnicas, para abordar el cálculo del flujo óptico, hayan sido propuestas. Pero todas las propuestas han tenido dos vertientes muy significativas y diferenciadas: la exactitud y la eficiencia para efectuar el cálculo del flujo óptico.

En Liu et al. [LHH+98] se desarrolló un trabajo donde se estudia la relación entre exactitud y eficiencia, sobre distintos tipos de algoritmos para calcular el flujo óptico. En el trabajo se presentó un cuadro comparativo, de varias técnicas para el cálculo de flujo óptico, donde se calcula el tiempo (en años) que debería pasar para que sea posible implementar en tiempo real dichos algoritmos. Los cálculos suponen que la capacidad computacional crece al doble cada año, dicha suposición es basada en la famosa ley de Moore presentada en 1965.

Los datos de Liu et al. arrojan información muy importante en la cual dice que algunos algoritmos pueden tardar hasta 14 años, considerando la fecha en realizar la tabla y suponiendo que la capacidad de cómputo crece al doble cada año. Por ejemplo, el algoritmo propuesto por Fleet [FJ90] tardaría 14 años para poder ser implementado en tiempo real, el propuesto por Horn [HS81] 12 años, el propuesto por Anandan [Ana89] 12 años, y el algoritmo que podría ser implementado en menos tiempo sería el propuesto por Lucas [LK81] que se tardaría 7 años.

Un punto importante es que por esas fechas, de la realización del trabajo de Liu et al., se pensaba que la capacidad de cómputo crecería al doble cada 12 meses. Hace pocos años cambió a cada 18 meses y en estudios recientes se cree que la tendencia seria menos favorable. En [Kis02] se habla del fin de la ley de Moore, argumentando que no existen limitaciones físicas, en lo que respecta al tamaño de los transistores, al menos durante 2 décadas. Pero que los efectos termodinámicos (ruido térmico) podrían causar un serio impacto sobre la integración de los circuitos integrados lo que a su vez limitaría las frecuencias de conmutación y limitar la capacidad de procesamiento de la información.

Por otro lado, recientes trabajos han demostrado que el cálculo de flujo óptico sigue teniendo un especial interés, tanto en obtener una alta precisión de los resultados como en cuestiones de eficiencia de procesado, que cae dentro del estudio del movimiento en tiempo real. Así varias arquitecturas se han propuesto para llevar a cabo el cálculo del flujo óptico en tiempo real, las cuales pueden ser clasificadas en tres grupos en función del dispositivo utilizado para su implementación. Estos grupos son: los que utilizan dispositivos de propósito general [Tag07] [FSVG07] [ALK07], dispositivos reconfigurables [DRP+06] [DRM+06] [MZC+05] y dispositivos de alta escala de integración diseñados a la medida (VLSI) [Sto06].

3.3.1. Arquitecturas con dispositivos de propósito general

Regularmente las arquitecturas de propósito general utilizadas para este propósito, el cálculo del flujo óptico, tienen que ser computadores paralelos o alguna arquitectura comercial disponible que posea una alta capacidad de cómputo. La razón es, como se ha dicho anteriormente, que los algoritmos existentes para el cálculo del flujo óptico demanda una elevada carga computacional además de una capacidad de almacenamiento considerable.

En [DCSN04] se presenta una arquitectura distribuida para el cálculo del flujo óptico. La arquitectura es un cluster con 8 nodos interconectados mediante una red Ethernet Gigabit. En el trabajo se implementa el algoritmo de Lucas y Kanade [LK81], logrando procesar 30 fps con imágenes de 502×288 píxeles, y sólo 10 fps con imágenes de 720×576 píxeles. También se especifica que la comunicación entre los nodos se lleva acabo mediante un sistema de pase de mensajes estándar, de código abierto, llamado LAM/MPI versión 6.5.8. de la Universidad de Indiana.

En [FCD01] se implementan tres algoritmos distintos: El algoritmo de Anandan utilizando una pirámide gausiana para su procesado [Ana89], el algoritmo de Fleet basado en la fase [FJ90] y un tercer algoritmo, el de Lucas y Kanade, basado en el gradiente [LK81]. Este último algoritmo es el que presentó mejores resultados en cuanto al tiempo de procesado y a la exactitud, según dicho trabajo. También se especifica que el tiempo de procesado es de $10,7$ s, con imágenes, de 252×316 píxeles, sobre una arquitectura de 4 procesadores en paralelo. Y se consigue una buena exactitud. El tiempo está un tanto alejado para un procesado en tiempo real, que como máximo debería ser de $41,66$ ms para lograr procesar 24 fps. Utilizando el mismo algoritmo del gradiente, el de Lucas y Kanade, en otro trabajo es implementado sobre una arquitectura miltiprocesador la MaxVideo200 [CC04]. En dicho trabajo sólo se indica que tiene una ganancia en tiempo respecto al trabajo anterior, el de [FCD01], pero también sufre una pequeña perdida de exactitud durante el procesado.

3.3.2. Arquitecturas con dispositivos reconfigurables

Este tipo de dispositivos son ideales para diseñar arquitecturas que requieran una flexibilidad de reprogramación y una excelente eficiencia de procesado. Por esa razón se han desarrollado en los últimos años distintas arquitecturas que utilizando este tipo de dispositivos realizan el cálculo de flujo óptico. Como se expuso en la

sección 2.2, existe una gran variedad de algoritmos para efectuar el cálculo del flujo óptico. Así pues, las diferentes arquitecturas desarrolladas hasta ahora con dispositivos lógicos programables, caen básicamente dentro de dos tipos de clases: la basada en la correlación [CM01] y la basada en el gradiente espacio temporal [Cob01] [CM03] [MRAE03] [MZC+05] y [DRP+06]. De esta última clase existen varios trabajos que utilizan algoritmos con distintos tipos de restricciones. Como son, el algoritmo de Lucas y Kanade [LK81], que considera restricciones locales y el algoritmo de Horn y Schunck [HS81], que considera restricciones globales.

Arquitectura que utiliza algoritmo de Camus

Esta arquitectura emplea un algoritmo correlacional. Este tipo de algoritmos es pocas veces utilizado para aplicaciones que requieran un procesamiento en tiempo real. La razón es el elevado coste computacional, como se explicó detalladamente en sección 2.2.2. Ted Camus, en [Cam94], presentó un trabajo basado en la correspondencia de regiones con el objetivo principal de procesar imágenes en tiempo real. En el trabajo propone una novedosa estrategia para mejorar los tiempos de procesado de las imágenes. La novedad de este algoritmo es que incorpora modificaciones en dos aspectos fundamentales, con la finalidad de reducir el tiempo empleado en la búsqueda de regiones similares en la secuencia de imágenes. La primera modificación implica el uso de un tiempo de muestreo variable de la imagen, lo que permite intercambiar el espacio con el tiempo. La segunda modificación permite extender la aplicación del algoritmo para crear un campo de flujo óptico con múltiples velocidades. Eso permite transformar las búsquedas espaciales cuadráticas en otras que sean lineales en el tiempo. Una explicación más detallada se puede ver en [Cam94].

En [CM01] se presenta la implementación hardware del algoritmo propuesto por Ted Camus. El algoritmo es del tipo correlacional con restricción del movimiento en la imagen. Es implementado sobre una FPGA, en particular la EPF10K50RC240-C de Altera. Se indica que logra procesar 22,56 fps, con imágenes con tamaño de 96×96 píxeles.

La arquitectura consiste de 5 bloques principales, como se puede observar en la figura 3.5. Los bloques de memoria actual y anterior son memorias RAM de doble puerto, externas a la FPGA utilizada. En las memorias se almacenan las imágenes que recibe de la cámara, del tipo CCD. Se accede a las imágenes conforme se vayan utilizando, mediante un circuito direccionador, representado por el bloque "Direccionador" dentro de la figura 3.5.

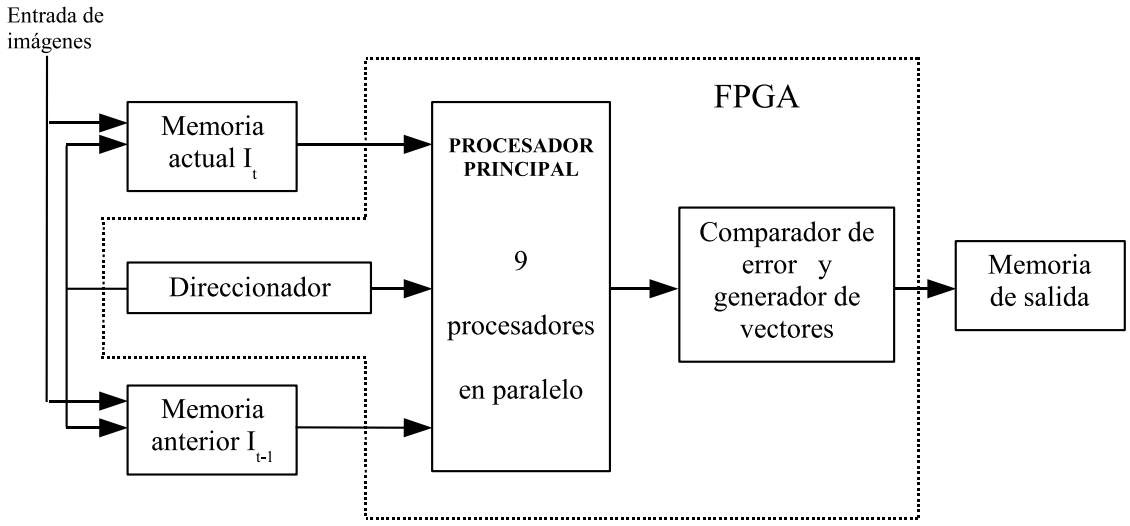


Figura 3.5: *Arquitectura para el cálculo del flujo óptico utilizada por Cobos en [CM01].*

El bloque direccionador se encuentra dentro de la FPGA y básicamente su función es generar la dirección y a la vez direccionar la memoria para acceder a los valores de cada uno de los píxeles de las imágenes, almacenadas previamente en la memoria. Los valores de los píxeles, de 8 bits, son entregados al bloque procesador.

En el bloque procesador, el más importante del sistema, se realizan las operaciones aritméticas del algoritmo. El objetivo del algoritmo es obtener un coeficiente de semejanza que será utilizado en la siguiente etapa del proceso (bloque comparador). El bloque procesador está constituido por 9 elementos de proceso o procesadores idénticos, los cuales se encargan de realizar el cálculo del coeficiente de semejanza de la posible región desplazada. Como se muestra en la figura 3.6. Se requieren 9 unidades de procesamiento para poder comparar simultáneamente la región de referencia respecto a las nueve posibles regiones en las que se puede encontrar en un instante de tiempo siguiente. Las ventanas de referencia son de 7×7 píxeles y las ventanas de búsqueda son de 9×9 píxeles, limitando el desplazamiento a un máximo de un píxel.

El bloque comparador de error y generador de vector analiza todos los coeficientes de semejanza de una región, detectando aquel que tenga el menor error y posteriormente genera el vector de flujo óptico correspondiente a cada píxel. La información será almacenada en una memoria de salida RAM de doble puerto.

La memoria de salida es un bloque externo a la FPGA. Al igual que las memorias

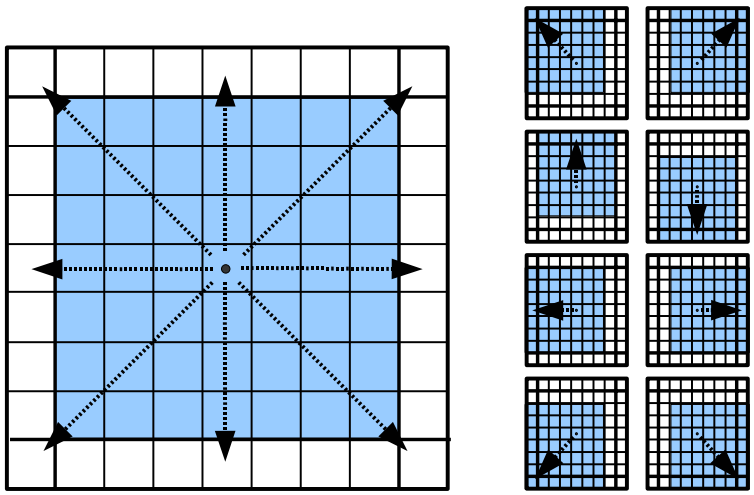


Figura 3.6: *Ventana de referencia de tamaño* 7×7 *píxeles sobre la ventana de búsqueda de* 9×9 *píxeles, mostrando sus 8 posibles desplazamientos o cuando no existe desplazamiento alguno.*

que conforman el *bloque actual* y el *bloque posterior*, el bloque memoria de salida son memorias RAM de doble puerto. Esto permite leer su contenido sin interferir en el proceso de escritura en la memoria. El tamaño de la memoria es de 32 Kbytes, permitiendo almacenar la información de flujo óptico de 3 imágenes consecutivas.

En el mismo trabajo se menciona que el dispositivo utilizado para capturar las imágenes es una cámara CCD B/N, con 256 tonos de gris y una resolución máxima de 480×480 píxeles. La imágen es submuestreada con un factor 5:1 para obtener imágenes con tamaño de 96×96 píxeles. Este submuestreo tiene como fin el poder procesar la información en tiempo real. Sin embargo, sólo pudieron procesar 22,56 fps.

El mismo autor propone una tapa adicional [Cob01]. La idea fundamental de dicha etapa es realizar una disminución de píxeles a procesar, pero sin llevar a cabo un submuestreo *general* de 5:1. Dicho de otro modo, sería procesar píxeles pero dando mayor importancia a la información de una área específica de la imagen. En el trabajo propone un sistema de visión en coordenadas log-polares, así la información excluida para el procesado sería sólo la de menor interés. La imagen logarítmico-polar que utiliza es de una resolución máxima de 96×128 píxeles. Lo que conllevaría a reducir el número de imágenes a procesar pues antes trabajaban con imágenes de 96×96 píxeles. Por otro lado requiere mayor cantidad de memoria que en el sistema propuesto inicialmente y una nueva etapa que requerirá recursos adicionales tanto

de hardware como en tiempo de ejecución para obtener las imágenes log-polares.

Arquitectura que utiliza el algoritmo de Lucas y Kanade

Una arquitectura que utiliza este tipo de algoritmo es propuesta por Díaz et al. en [DRP+06]. El sistema es implementado sobre una tarjeta de desarrollo RC1000-PP de Celoxica, utilizando Handel C como el lenguaje de descripción de hardware. La placa se conecta al computador a través del bus PCI. La tarjeta de desarrollo contiene como componente principal la FPGA Virtex-E XCVE2000-4 y 4 bancos de memoria SRAM de 2 MBytes cada uno, accesible en paralelo.

El sistema diseñado se denomina *sensor virtual* y consiste de una cámara convencional como sensor de entrada y un elemento de proceso, que es implementado sobre una FPGA. El elemento de proceso consiste de un frame-grabber, un circuito de estimación del flujo óptico y la lógica de control necesaria para el sistema.

El diseño utiliza al máximo las posibilidades de paralelismo y segmentación de cauce, que son de las principales ventajas que proporcionan los dispositivos lógicos programables como las FPGA's, aparte de su capacidad de reprogramación. En este caso la arquitectura del sistema consiste de un cauce segmentado compuesto por 6 etapas, como se muestra en la figura 3.7.

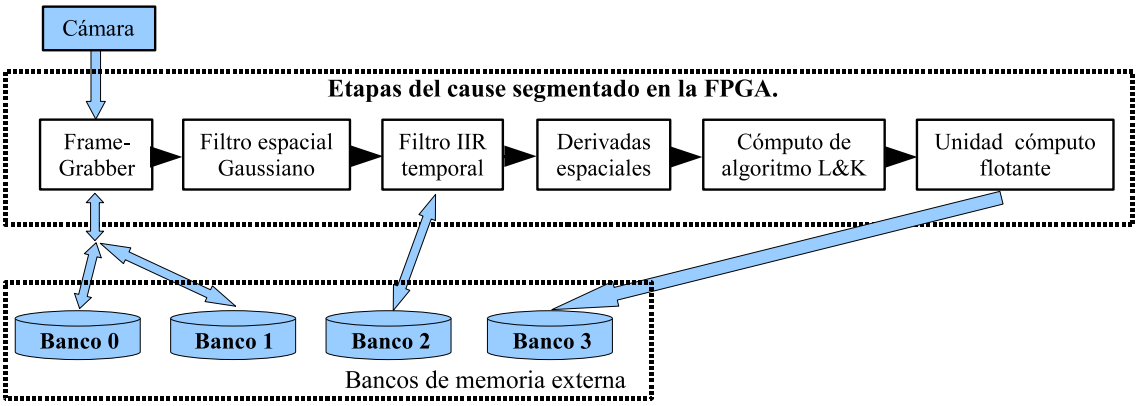


Figura 3.7: *Arquitectura para el cálculo del flujo óptico, utilizando el algoritmo de Lucas y Kanade, propuesta por Díaz et al. en [DRP+06].*

La primera etapa, el *frame-grabber*, recibe las imágenes de la cámara que serán almacenadas en dos bancos de la memoria SRAM externa. En la segunda etapa se realiza el proceso de suavizado espacial de la imagen con filtros Gaussianos. En la tercera etapa se aplica un procesamiento temporal de la imagen con filtros IIR

para el suavizado y cálculo de la derivada. El filtro temporal IIR necesita almacenar 3 imágenes anteriores. Debido a que se requiere almacenar valores temporales en está etapa, se utiliza un tercer banco de memoria SRAM (Banco 2). En la cuarta etapa se realiza la estimación espacial de las derivadas vertical y horizontal de la imagen. En la quinta etapa del cause se realiza el cómputo del flujo óptico, utilizando el algoritmo de Lucas y Kanade, que consiste básicamente de un conjunto de operaciones matriciales. En la última etapa, que es una unidad procesadora de punto flotante ayuda para la realización de la estimación final de la velocidad. Una vez calculada la velocidad los datos son almacenados en un cuarto banco de memoria SRAM de doble puerto (Banco 3).

El trabajo concluye que el sistema es capaz de procesar 24 fps de 320×240 píxeles. Las imágenes son de 256 tonos de gris (8 bits).

Arquitectura que utiliza el algoritmo de Horn y Schunck

Este tipo de algoritmo es el más conocido y estudiado de la literatura. Por otro lado es el que presenta una mayor densidad de flujo óptico, respecto a los dos algoritmos utilizados en las arquitecturas anteriores. Cualquier arquitectura que utilice este tipo de algoritmo deberá contar con cinco bloques particulares. Una memoria de entrada que deberá almacenar dos imágenes consecutivas, otra memoria para almacenar los valores resultantes, un elemento que ejecute el cálculo del gradiente espacio-temporal, otro que realice el cálculo del flujo óptico y un último bloque que realice la Laplaciana.

En [Cob01] se presenta una arquitectura que implementa este algoritmo. Utiliza dos FPGA's, la 4020EHQ208-3 y la 4005HP6223-5, de la familia XC4000 de Xilinx. La arquitectura cosiste de 6 bloques, como se muestra en la figura 3.8.

1. **Memoria de entrada.** Almacena dos imágenes consecutivas. Este componente es externo a cualquier dispositivo FPGA utilizado en el sistema. La memoria es del tipo SRAM con un tiempo de acceso de 40 ns.
2. **Bloque principal.** Controla las iteraciones y temporiza el inicio y el fin del procesado de las imágenes. También genera las direcciones del píxel que se va a procesar utilizando un contador de filas y columnas.
3. **Bloque gradiente.** Este bloque realiza el cálculo de los gradientes espacio-temporal (I_x, I_y e I_t), utilizando las dos imágenes consecutivas almacenadas en la memoria de entrada.

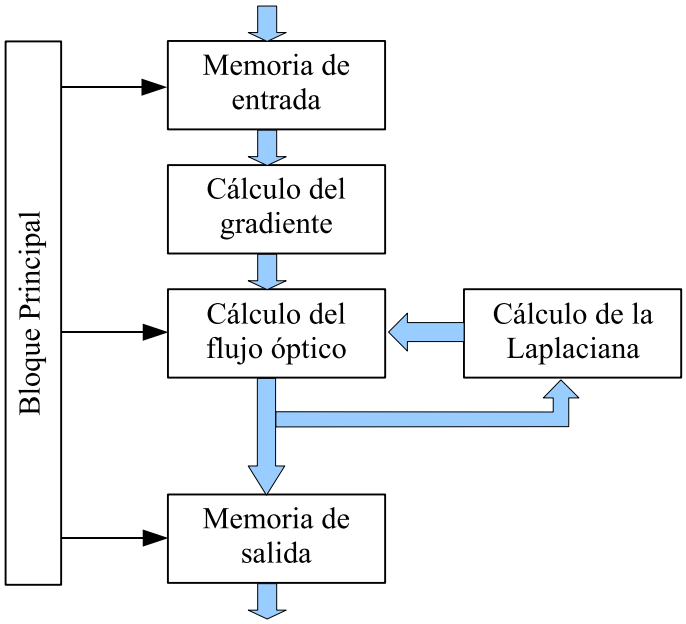


Figura 3.8: *Arquitectura para el cálculo del flujo óptico, utilizando el algoritmo de Horn y Schunck, propuesto por Cobos en [Cob01].*

4. **Bloque Laplaciana.** Aquí se realiza el suavizado global, que consiste en utilizar una máscara sobre los valores obtenidos del flujo óptico. La máscara requiere los 8 píxeles adyacentes al píxel que se le está aplicando la Laplaciana.
5. **Bloque flujo óptico.** En este bloque se realiza el cálculo del flujo óptico. Esta ecuación es iterativa y depende de los valores de salida de la Laplaciana. Así que los valores iniciales para el primer cálculo serán cero.
6. **Memoria de salida.** Se utiliza para almacenar los valores resultantes del flujo óptico.

Los bloques 2 y 3 son implementados en la FPGA 4005HP6223-5 y los bloques 4 y 5 son implementados en una 4020EHQ208-3. Las memorias RAM de entrada y de salida son dispositivos externos a cualquier FPGA. El sistema procesa 19 fps, con imágenes de 50×50 píxeles de 8 bits y con un número de iteraciones constante, igual a 3 por cada par de imágenes consecutivas.

En [MZC+05], también es implementado el algoritmo de Horn y Schunck. En ese trabajo utilizan una sola FPGA, la EP20K300EQC240-2, y procesan hasta 60 fps, con imágenes de 256×256 píxeles de 8 bits, pero con sólo una iteración por par de imágenes. Es importante notar que la ecuación de Horn y Schunck tiene la particularidad de que es una función ***iterativa***. Entonces la diferencia de esta arquitectura,

respecto a la antes expuesta, radica principalmente en utilizar la salida de la primera iteración como entrada para el cálculo del flujo óptico de un nuevo par de imágenes. Esto se repetirá hasta haber procesado 64 pares de imágenes y una vez hecho eso se presentará en la salida el valor del flujo óptico. De esta forma las iteraciones son hechas entre 64 imágenes consecutivas y no sólo entre dos imágenes, como el caso anteriormente expuesto que hacia 3 iteraciones por cada par de imágenes. Debido a que se excluye la realización iterativa entre un mismo par de imágenes, se hace innecesario un circuito de control de las iteraciones y éste es sustituido por un simple contador de 0 a 63. Esto y la elección de memorias que adecúan la información contribuye a que la arquitectura posea un alto grado de segmentación y permite mejorar los tiempos de ejecución y procesamiento de la información . El diagrama a bloques, para este tipo de arquitectura, es como el mostrado en la figura 3.9

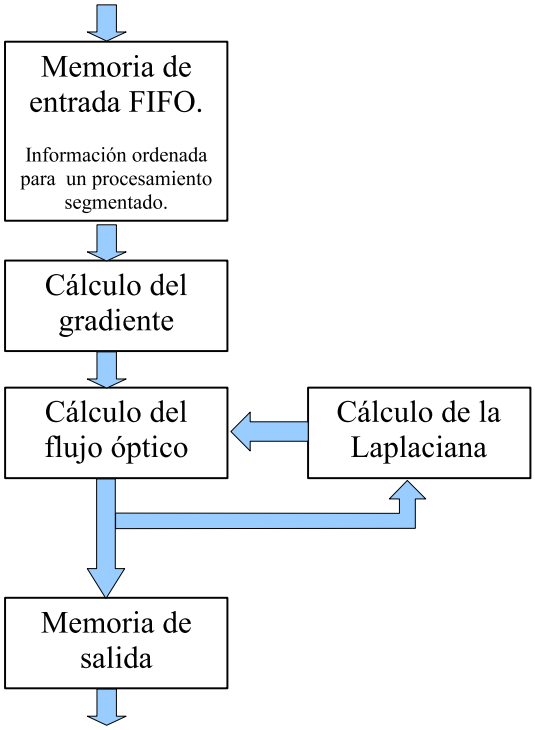


Figura 3.9: *Arquitectura para el cálculo del flujo óptico, utilizando el algoritmo de Horn y Schunck, propuesto por Martín et al. en [MZC+05].*

La memoria utilizada es una FIFO IDT7208 y son externas a la FPGA. Los datos contenidos en la memoria están organizados de forma tal que con cierto desplazamiento entre renglones sirve para procesar eficientemente la información. Esto permite que el procesamiento de datos de las diferentes etapas sea llevado a cabo en paralelo, todo ello gracias al flujo de datos que depende mucho de cómo se encuen-

tran almacenados en la memoria, que es un píxel después de otro. De esta forma mientras se está calculando un bloque del sistema, como sería el procesado del flujo óptico de un píxel, otro bloque estaría realizando simultáneamente el cálculo del gradiente espacio-temporal para el siguiente píxel. El resultado de cada bloque fluye al siguiente bloque a la misma velocidad que entra un nuevo píxel al sistema de procesado.

3.3.3. Arquitecturas con dispositivos dedicados VLSI

Este tipo de arquitecturas sería la forma más eficiente en cuanto al procesado se refiere, pues se podrían obtener altas frecuencias de procesado y se llevarían a cabo los procesos al límite de paralelización que permite el algoritmo a implementar. Sin embargo, los inconvenientes son varios. El primero de ellos es el alto coste del dispositivo o de la fabricación. Otro sería el tiempo requerido de diseño y de fabricación del dispositivo aumentando así el tiempo de diseño del sistema total. Otro más de sus inconvenientes es que actualmente existen solo dispositivos prototipos con una limitada resolución. Un último inconveniente es que cuando se utiliza para diseñar nuevos sistemas prototipos las arquitecturas que trabajan con estos dispositivos pierden flexibilidad. La razón es por el hecho de que no pueden ser reconfigurados o reprogramados por necesidades del sistema, como podía ser el tener una mayor resolución, a menos que se diseñe y fabrique un nuevo dispositivo.

Actualmente existen trabajos que han utilizado sensores de 15×15 píxeles [SD04]. En un trabajo más reciente se utilizó un sensor de 30×30 píxeles [Sto06]. En ambos trabajos se utilizó tecnología BiCMOS de $0{,}8\mu m$, doble metal, doble polisilicio. En el último trabajo se especifíca que posee una frecuencia de muestreo de 67 fps. Además, según el documento, podrían alcanzar mucho mayores frecuencia de muestreo-procesado, algo que ninguna otra arquitectura podría proporcionar hoy en día.

Sin lugar a duda sus tres principales ventajas nunca se podrán discutir, una alta inmejorable capacidad de cómputo, algo que para el procesado de imágenes en tiempo real es indispensable. Una segunda ventaja es el bajo consumo de potencia que requeriría tanto para capturar y obtener el flujo óptico de las imágenes capturadas. La tercera ventaja son las dimensiones físicas, las cuales son muy reducidas. Como se puede observar en la figura 3.10[5], en la que muestra una comparación entre las

[5]http://www.cns.nyu.edu/ alan/research

dimensiones del sensor y una moneda de 10 centavos de dólar.

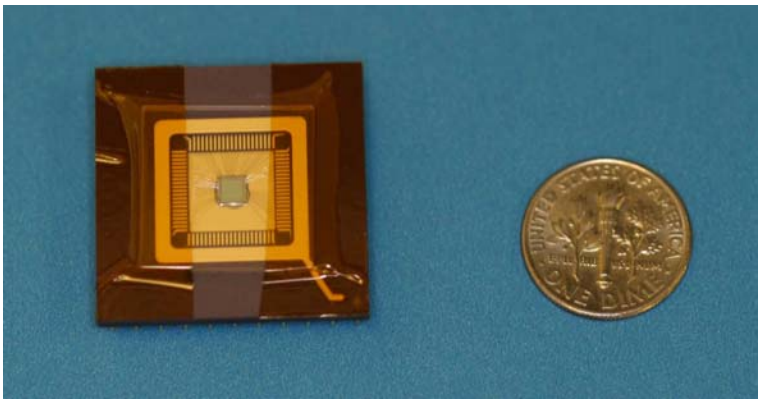


Figura 3.10: *Sensor de flujo óptico 2D, con una resolución de* 30×30 *píxeles y una moneda de 10 centavos de dólar como referencia.*

Así pues, este tipo de arquitectura es ideal para sistemas que requieran una alta frecuencia de procesado, pero que no requieran una alta resolución en las imágenes.

Con esto es posible concluir que, para diseñar sistemas que permitan procesar imágenes en tiempo real que permitan cierta flexibilidad de reprogramación y que las arquitecturas sean de dimensiones aceptables, sólo se cuenta con la alternativa de los dispositivos reconfigurables. Esta alternativa es la más indicada por las dimensiones físicas de los dispositivos y por el equilibrio de diseño que guarda, al tener una flexibilidad software y una eficiencia hardware.

Parte II

Diseño y programación del sistema

Capítulo 4

El procesado de imágenes guiado por cambios

4.1. Introducción: Principio básico

En los sistemas de visión artificial el método clásico de procesamiento de imágenes comúnmente realiza el procesado total de cada imagen de una secuencia. Es decir, se procesa cada uno de los píxeles que constituye la imagen y esto se repite para cada imagen de la secuencia. Este proceso se hace incluso si dentro de una secuencia de imágenes existen pares de imágenes consecutivas idénticas. Además, es común que los sistemas de procesamiento de imágenes consideren que la intensidad en cada píxel de la imagen es constante en el tiempo. Esto indica que las variaciones de intensidad en la imagen se deberán única y exclusivamente a desplazamientos de objetos dentro de la escena y no a cambios de intensidad.

Por ejemplo, en el caso de los sistemas de reconocimiento y seguimiento de objetos, el procedimiento normal para el procesado implica la aplicación de un algoritmo de reconocimiento para cada imagen involucrada en la secuencia de imágenes. Posteriormente, mediante otro algoritmo se realizaría el seguimiento del objeto en la misma secuencia de imágenes. Todo este proceso, dependiendo de la complejidad de los algoritmos y la resolución de las imágenes, podría llevar mucho tiempo, incluso varios segundos, dificultando la ejecución del sistema en tiempo real.

Una alternativa para reducir el tiempo de procesado es lograr detectar la información considerada importante de la escena, dentro de una secuencia de imágenes, y procesar sólo la información de interés. Esto es posible si se tiene en cuenta que

regularmente son pocos los cambios entre dos imágenes consecutivas, especialmente si el tiempo de adquisición entre ambas imágenes es pequeño. Entonces si se logra detectar sólo los píxeles que han cambiado, dentro de una secuencia de imágenes, y procesar esa información es posible reducir el tiempo de cálculo del sistema. Además, si también se considera que en una secuencia de imágenes pueden existir dos o más pares de imágenes consecutivas sin que se registre cambio alguno es posible asegurar con toda certeza una reducción del tiempo total de cálculo evitando el procesado de los pares de imágenes que no cambian.

De esta manera, si sólo son procesados los datos de importancia y no toda la imagen, como se hace típicamente, y si se desecha la imagen consecutiva que no presente cambio alguno, es posible afirmar con una alta certeza que se reducirá drásticamente el tiempo requerido por el sistema para el procesado total de la información. El objetivo primordial del presente trabajo de investigación es disminuir el tiempo de procesado de la información para procesar información en tiempo real.

El principio de considerar sólo el movimiento que existe en una escena no es nuevo. Durante el estudio del sistema de visión humano se ha constatado que existe un principio similar que indica la existencia de una relación muy estrecha entre la información procesada por el cerebro humano y el movimiento percibido por el sistema visual [WMVB94].

El ojo humano puede funcionar sobre un gran rango de iluminación y también es capaz de soportar diferentes cambios de intensidad de la iluminación. Constantemente el ojo captura la información proyectada periódicamente en forma de imagen en la retina. La información es integrada de tal forma que los objetos de la escena aparecen estáticos o con movimientos suaves. Debido a que el tiempo empleado para recibir y procesar la información es finito, entonces el sistema de visión es más **sensible** sólo a los cambios percibidos en la nueva adquisición de la información. Además de esto, el sistema de visión humano es capaz de diferenciar entre un cambio de intensidad de iluminación y un movimiento en la escena. En 1958 De Lange caracterizó una función denominada *función de sensibilidad de contraste temporal* (FSCT). En 1994 Metha et al. [WMVB94], realizó un trabajo donde se indica que existen diferentes sensibilidades, para los diferentes componentes del sistema de visión, con el objetivo primordial de discriminar el movimiento en la escena. De esta manera, para la discriminación del movimiento, detección e identificación se proporcionan ciertos umbrales.

De todo esto se entiende que para poder llevar a cabo el procesado guiado por

cambios, propuesto en este trabajo, es necesario contar con una función que determine cuándo se considera que un píxel ha cambiado en función a su contraste y entonces se inicie su procesado. Esta función de sensibilidad de contraste (FSC), determinará que cuando la magnitud de la diferencia de los dos mismo píxeles de dos imágenes consecutivas supera un umbral establecido (t_h), entonces existirá información relevante en ese píxel o un cambio ($C_h = 1$), iniciando el procesado del píxel. En otro caso, si la magnitud es menor o igual al umbral (t_h), se considera que no existe cambio alguno ($C_h = 0$) y por lo tanto dicho píxel no será procesado. Esta función se puede escribir como:

$$C_h = \begin{cases} 1 & \text{si} \quad mag > t_h \\ 0 & \text{si} \quad mag \leq t_h \end{cases} \tag{4.1}$$

La selección del umbral (t_h) es sumamente importante y se verá detalladamente en el capítulo 5. La razón es que un valor de umbral muy pequeño puede ocasionar que cualquier tipo de cambio de iluminación en un píxel por pequeño que sea, ya sea inducido por la cámara en movimiento, ruido del sensor o pequeñas variaciones de iluminación, pueda dejar sin efecto esta técnica. Eso podría provocar que se procese toda la imagen como tradicionalmente se hace, agregando además un tiempo extra en el proceso de detección de cambios, resultando contraproducente sobre todo si se implementa secuencialmente. Por otro lado valores muy grandes de umbral pueden eliminar información importante al no considerar cambios en la escena debido a pequeños movimientos o a movimiento entre dos objetos que tengan una pequeña diferencia de color y ésta se encuentre dentro del umbral del contraste.

Un diagrama de bloques que representaría este principio, en conjunto con el sistema de procesado, es mostrado en la figura 4.1. En la misma figura se puede ver que son requeridas dos imágenes consecutivas (Im_{t-1} *e* Im_t). Una vez que se capturan dos imágenes consecutivas arranca el proceso para la detección de los cambios y su posterior procesado. Este hecho se repetirá sucesivamente sustituyendo la última imagen capturada (Im_t) por una nueva imagen. Simultáneamente la imagen actual (Im_t) sustituiría a la imagen de referencia (Im_{t-1}). En realidad lo único que cambia sería un apuntador que direcciona a la imagen actual y la de referencia. También se puede apreciar que la etapa de procesamiento de imágenes es independiente del bloque que realiza la detección de los cambios entre las dos imágenes consecutivas. Sin embargo, dependiendo del tipo de algoritmo utilizado para el procesamiento puede o no incluirse en el mismo algoritmo. Es posible considerar que con una pequeña modifi-

cación se realicen simultáneamente ambos procesos, tanto la detección de cambios y el procesado de los píxeles. Se aclara esto debido a que se podría creer que siempre, al incorporar el procesado guiado por cambios, es agregado un tiempo adicional al requerido por el algoritmo para el procesado. La variable $LUT_{(i,j)}$ es una tabla que almacena la posición de los píxeles que han cambiado y que posteriormente serán procesados.

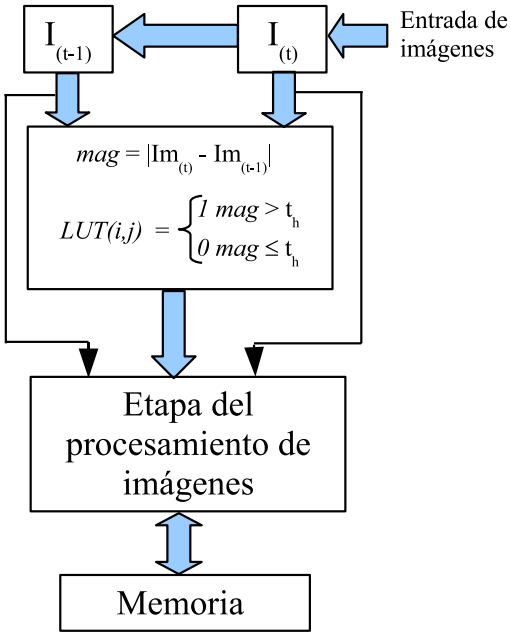


Figura 4.1: *Diagrama general a bloques de un sistema de procesamiento de imágenes incorporando el procesado guiado por cambios.*

4.1.1. Análisis general del procesado guiado por cambios

La propuesta de este trabajo es procesar sólo los datos que han cambiado entre una imagen actual y una imagen de referencia, capturada en un instante de tiempo anterior (Δ_t), descartando su procesamiento cuando las imágenes son idénticas. Típicamente, en los sistemas de procesamiento de imágenes, se realiza el procesado total de la imagen y de cada una de las imágenes que conforman la secuencia. El hecho de procesar todas las imágenes en su totalidad es más costoso, que el procesar sólo los píxeles que han presentado cambios significativos (por arriba de un umbral establecido).

Existen varios factores que al utilizar la técnica del procesado guiado por cambios

inciden directamente en la reducción del número de ciclos para el procesado de la información. Dos de los factores más relevantes son: la disminución de datos a procesar y la disminución de accesos a memoria. Por otro lado, con un diseño orientado en la segmentación de la arquitectura se puede reducir aun más el tiempo del procesado total.

Disminución de datos a procesar

Al capturar una nueva imagen y compararla con una imagen de referencia, capturada previamente con un pequeño intervalo de tiempo, es posible detectar, localizar y procesar sólo los píxeles que han presentado cambios significativos por arriba de un umbral previamente establecido. Para llevar a cabo este propósito es necesario incorporar un bloque adicional o bien realizar una pequeña modificación sobre el algoritmo empleado por el sistema, con el objetivo de localizar y separar los píxeles que han presentado cambios significativos para su futuro procesado. Al reducirse el número de píxeles a procesar es posible reducir el número total de ciclos necesarios para el procesado de las imágenes.

Sea Φ el número de ciclos necesarios para el procesado de un píxel de una imagen de tamaño $m \times n$, utilizando un algoritmo específico y suponiendo un procesamiento secuencial. Entonces el número de ciclos necesarios para procesar toda la imagen está dado por:

$$\Phi_T = \Phi(m \times n) \tag{4.2}$$

Bajo las mismas condiciones anteriores, para un procesado guiado por cambios, la ecuación estaría representada por:

$$\Phi_{TGC} = \Phi(m \times n) \cdot (1 - \frac{\eta}{m \times n}) \tag{4.3}$$

donde η es el número de píxeles que no cambiaron, no mostraron cambios significativos, que hayan superado un umbral establecido, o que se mantuvieron constantes entre la imagen de referencia (Im_{t-1}) y la imagen actual (Im_t). Esos píxeles no serán procesados. Cuando se de el caso en que las imágenes consecutivas sean completamente diferentes, siendo un caso poco probable, entonces $\eta = 0$ haciendo que $\Phi_{TGC} = \Phi_T$. El caso opuesto cuando $\eta = m \times n$, significando que ambas imágenes consecutivas son idénticas, entonces se tiene que el número de ciclos necesarios para el procesado sería $\Phi_{TGC} = 0$.

Es importante mencionar que el valor de η está íntimamente ligado con el valor asignado al umbral (t_h), es decir para valores altos de t_h el valor de η se incrementará.

Disminución de accesos a memoria

El pretender reducir el número de ciclos necesarios para realizar en menor tiempo el procesado de la información está directamente relacionado con los accesos a la memoria, tanto para leer como para escribir los píxeles de la imagen. Al reducir el número de veces para leer/escribir un píxel a procesar/procesado se ve reflejado en una reducción del tiempo total necesario para el procesado.

Cada dispositivo de memoria tiene un tiempo de acceso t_{acc} según el tipo de memoria utilizada por el sistema. Al considerar un bus de datos de 32 bits y que cada píxel es de 8 bits, entonces es posible leer/escribir 4 píxeles por cada acceso a memoria. Ahora bien, si por cada píxel que se lee es necesario escribir su resultado entonces se puede decir que al menos se duplicaría el número de acceso a memoria por cada píxel a procesar. La función que represente el tiempo de acceso a memoria para todos los píxeles de la imagen puede ser:

$$t_{accT} = 2\, t_{acc} \frac{m \times n}{4} = t_{acc} \frac{m \times n}{2} \tag{4.4}$$

Para un procesado guiado por cambios la eciación estaría representada por:

$$t_{accTGC} = t_{acc}\, \frac{m \times n}{2} \cdot (1 - \frac{\eta}{m \times n}) \tag{4.5}$$

Segmentación de la arquitectura

Una estrategia para reducir el número de ciclos necesarios en el procesado de la información es aumentando la eficiencia de la arquitectura mediante la segmentación de la arquitectura. Comúnmente las arquitecturas clásicas para el procesamiento de imágenes realizan el procesado de la información de forma secuencial. Esto es, primero se accede a la memoria por el dato o datos a procesar, posteriormente se ejecuta y se procesa la información utilizando un algoritmo específico. Una vez procesada la información el dato resultante es almacenado en la memoria de salida. Este procedimiento se realiza un número de veces como el número de píxeles hayan presentado cambios significativos en las imágenes.

Cuando existe la posibilidad de diseñar una arquitectura en función de las necesidades o de las características específicas del algoritmo a implementar, como es el caso de este trabajo que utilizará dispositivos lógicos programables, es importante conocer la viabilidad para implementar una arquitectura segmentada. La finalidad de un cause segmentado es para hacer más eficiente la arquitectura y aprovechar al máximo la flexibilidad de los dispositivos utilizados. Así mientras una parte de la arquitectura realiza una operación sobre un píxel, otra parte de la arquitectura realizará una operación diferente al píxel que lo precede. Esto es posible de realizar si el flujo de datos es secuencial, un píxel después de otro píxel.

En las arquitecturas con un cauce segmentado la velocidad del flujo de datos viene determinada por la etapa más lenta del proceso. Por lo tanto la división del algoritmo en varias etapas debe ser equilibrada en cuanto al coste temporal para evitar cuellos de botella. Por otro lado dado que siempre se desconocerá el número de píxeles que cambian entre los pares de imágenes consecutivas se tiene que la temporización del intercambio de los datos no es fija y se debe adaptar a las operaciones de las diferentes etapas en que se dividirá el algoritmo o proceso.

Para llevar a cabo un análisis detallado de la segmentación de la arquitectura, es necesario conocer el algoritmo específico utilizado en el procesamiento de las imágenes. Después se deberá definir un protocolo de intercambio de datos y por lo tanto unas señales de control.

4.2. Análisis del cálculo del flujo óptico guiado por cambios

Antes de realizar un análisis detallado del procesado guiado por cambios es necesario definir el algoritmo específico para llevar a cabo el cálculo de flujo óptico. El algoritmo utilizado en este trabajo, para el cálculo del flujo óptico, es el de Horn y Schunck, visto anteriormente en la sección 2.2. Se eligió este algoritmo por tres razones. La primera de ellas es el hecho de que proporciona una densidad de flujo óptico del 100 %, de la imagen. Algo que no todos los algoritmos hacen y particularmente ninguno del resto de los algoritmos vistos en la sección 3.3.2. La segunda razón es por la complejidad y el alto costo computacional que representa este algoritmo. Finalmente una tercera razón es por el hecho de que este algoritmo es muy buen punto de referencia, ya que es uno de los algoritmos más estudiados en la literatura y sigue siendo utilizado [DDB04] [MM04] [BW05] y [MZC+05].

Para realizar la implementación del algoritmo de flujo óptico guiado por cambios (FOGC) una primera aproximación consiste en la utilización de la arquitectura clásica. Esto es, implementando el algoritmo por software y programando en C++. La idea básica al utilizar esta primera aproximación, fue conocer la viabilidad de paralelismo del algoritmo, alternativas para la segmentación del procesado y evaluar la capacidad computacional de cada operación, que interviene en el algoritmo, con la idea de su futura implementación en hardware. También, sirve para evaluar los beneficios y el rendimiento de la estrategia propuesta en este trabajo. Una ventaja adicional de la implementación en la arquitectura clásica, que sin lugar a duda es significativa, consiste en la capacidad de monitorizar rápidamente el resultado (desplegado de las imágenes), algo que es relativamente más fácil.

A continuación se describirá brevemente el algoritmo utilizado para el cálculo del flujo óptico, algoritmo explicado detalladamente en la sección 2.2.1. Las ecuaciones 2.19 y 2.20 son reescritas de la forma:

$$u^{n+1} = \overline{u}^n - I_x \frac{I_x \overline{u}^n + I_y \overline{v}^n + I_t}{\lambda^2 + I_x^2 + I_y^2} \tag{4.6}$$

$$v^{n+1} = \overline{v}^n - I_y \frac{I_x \overline{u}^n + I_y \overline{v}^n + I_t}{\lambda^2 + I_x^2 + I_y^2} \tag{4.7}$$

donde I_x, I_y e I_t son las derivadas parciales, $\overline{u}$ y $\overline{v}$ son los coeficientes promedio de la velocidad (u,v) y λ^2 es una constante de la restricción de suavidad. Se puede notar que las ecuaciones muestran una dependencia espacial iterativa con la velocidad. Entonces $\overline{u}$ y $\overline{v}$ se calculan mediante:

$$\overline{u}_{i,j} = \frac{1}{6}(u_{i-1,j} + u_{i,j+1} + u_{i+1,j} + u_{i,j+1}) + \frac{1}{12}(u_{i-1,j-1} + u_{i-1,j+1} + u_{i+1,j+1} + u_{i+1,j-1}), \tag{4.8}$$

$$\overline{v}_{i,j} = \frac{1}{6}(v_{i-1,j} + v_{i,j+1} + v_{i+1,j} + v_{i,j+1}) + \frac{1}{12}(v_{i-1,j-1} + v_{i-1,j+1} + v_{i+1,j+1} + v_{i+1,j-1}), \tag{4.9}$$

Retomando las ecuaciones 4.6 y 4.7, es posible apreciar que falta llevar a cabo el cálculo de las derivadas parciales, I_x, I_y e I_t, para poder obtener los valores de la velocidad del flujo óptico. Una aproximación para el cálculo de las derivadas parciales sería la media de la diferencia de tono de gris utilizando una máscara 3D (x, y, t) de 8 píxeles de tamaño 2×2 por cada imagen, de las dos imágenes consecutivas, siendo definidas por:

$$I_x \approx \frac{1}{4}\{I_{i,j+1,k} - I_{i,j,k} + I_{i+1,j+1,k} - I_{i+1,j,k} + I_{i,j+1,k+1} - I_{i,j,k+1} + I_{i+1,j+1,k+1} - I_{i+1,j,k+1}\} \tag{4.10}$$

$$I_y \approx \frac{1}{4}\{I_{i+1,j,k} - I_{i,j,k} + I_{i+1,j+1,k} - I_{i,j+1,k} + I_{i+1,j,k+1} - I_{i,j,k+1} + I_{i+1,j+1,k+1} - I_{i,j+1,k+1}\} \tag{4.11}$$

$$I_t \approx \frac{1}{4}\{I_{i,j,k+1} - I_{i,j,k} + I_{i+1,j,k+1} - I_{i+1,j,k} + I_{i,j+1,k+1} - I_{i,j+1,k} + I_{i+1,j+1,k+1} - I_{i+1,j+1,k}\} \tag{4.12}$$

Con esta última aproximación es posible iniciar el proceso del cálculo del flujo óptico. Es importante resaltar que los valores iniciales del promedio de la velocidad $\overline{u}$ y $\overline{v}$ serán cero en la primer iteración.

Los diagramas a bloques que representa la secuencia de ejecución de las etapas del algoritmo para el procesado del FO y pare el procesado del FOGC, son mostrados en la figura 4.2. Se puede observar claramente que la diferencia existente entre estos dos diagramas a bloques radica en el módulo que detectará y ubicará los píxeles que han cambiado significativamente. El objetivo del módulo es saber la localización de los píxeles, que presentaron cambios significativos, creando una tabla (LUT) o imagen que contendrá la dirección de dichos píxeles.

4.2.1. Implementación directa e implementación guiada por cambios, para el cálculo del flujo óptico

La implementación directa consiste en una transcripción literal del diagrama a bloques de la figura 4.2 (izquierda) a su equivalente en código de programación C++. La implementación, para efectuar el cálculo del flujo óptico, se puede resumir en 5 pasos o etapas.

1. Capturar y almacenar dos imágenes consecutivas en la memoria local del sistema.
2. Calcular los gradientes espacio-temporal (I_x, I_y e I_t).
3. Calcular la velocidad (u, v) considerando a $\overline{u}$ y $\overline{v}$ iniciales igual a cero.
4. Calcular $\overline{u}$ y $\overline{v}$ y repetir el paso 4 un número k de veces, k representa el número de iteraciones y es establecido por el usuario.
5. Si ya se efectuaron las k iteraciones, entonces se presenta en la salida los valores de la velocidad (u, v).

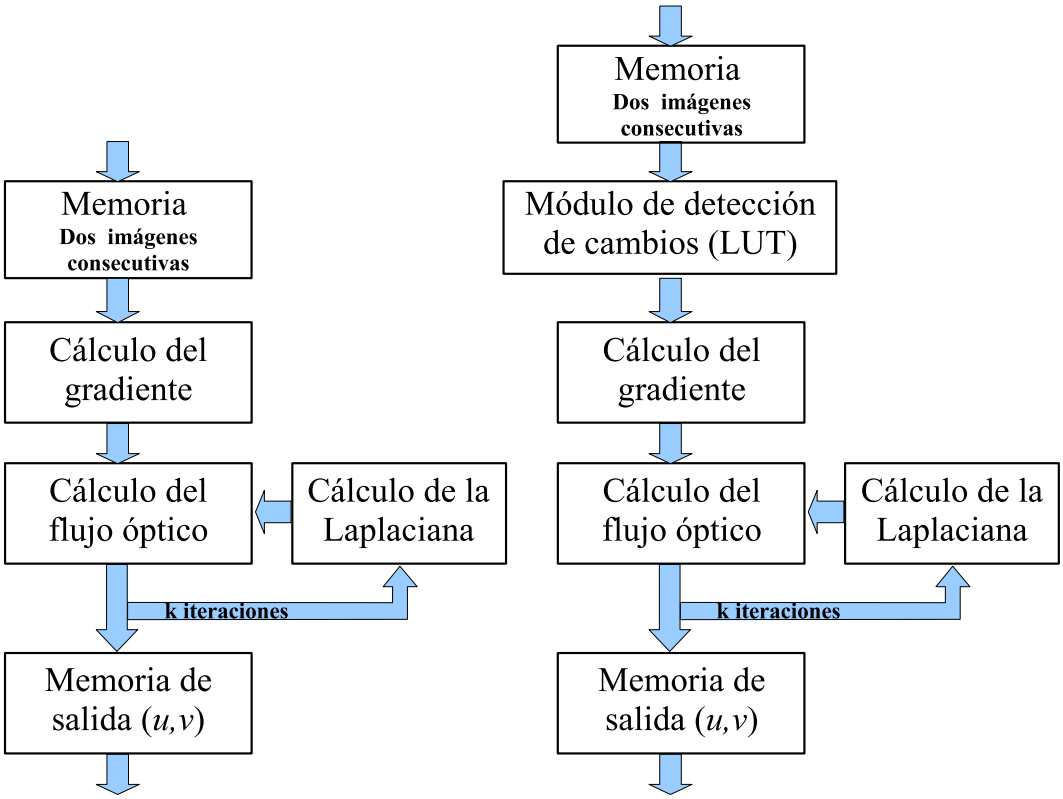


Figura 4.2: *Diagrama a bloques comparativo en el que se muestran las etapas para efectuar el procesado del flujo óptico con el procesado guiado por cambios (derecha) y la foma típica, utilizando el algoritmo de Horn y Schunck.*

Para que el algoritmo realice todo este proceso se requiere un determinado número de ciclos, que viene dado por:

$$ciclos = 8\,(m \times n) + k(16\,(m \times n) + 5\,(m \times n)) \tag{4.13}$$

El primer término representa la aproximación del número de ciclos necesarios para llevar a cabo el cálculo del gradiente. El segundo término especifica el número de ciclos para realizar el cálculo de la velocidad y de la Laplaciana multiplicados por un factor k, la cual representa el número de iteraciones. Las constantes 8 16 y 5 es en función de los términos requeridos por la función que se ejecute. Por ejemplo, el 8 se debe a que se requieren 8 píxeles por cada píxel que se requiera procesar para obtener su gradiente.

Ahora bien, para el caso del procesado del flujo óptico guiado por cambios es posible resumirlo en 6 pasos o etapas:

1. Capturar y almacenar dos imágenes consecutivas en la memoria local del sistema.
2. Crear una tabla (LUT (i, j)) que apuntará a los píxeles que presentaron cambios significativos entre las dos imágenes consecutivas.
3. Calcular los gradientes espacio-temporal I_x, I_y e I_t.
4. Calcular la velocidad (u, v), considerando a $\overline{u}$ y $\overline{v}$ iniciales igual a cero.
5. Calcular $\overline{u}$ y $\overline{v}$ y repetir el paso 3 un número k de veces. k representa el número de iteraciones y es establecido por el usuario.
6. Si ya se efectuaron las k iteraciones, entonces se presenta en la salida los valores de la velocidad (u, v)

Se puede notar que la diferencia es sólo un paso y consiste en la obtención de la $LUT(i, j)$. En esa etapa se realizan dos accesos de lectura a memoria por cada píxel del par de imágenes consecutivas. Y se realiza la diferencia entre los dos píxeles $|Im_{t-1}(i, j) - Im_t(i, j)|$. Si la diferencia es mayor a un umbral previamente establecido entonces se realiza una escritura a memoria para almacenar la posición (i, j) del píxel.

El número de ciclos de reloj necesarios para llevar a cabo el algoritmo de flujo óptico guiado por cambios estaría representado por:

$$ciclos = 3\,(m \times n) + 8\rho\,(m \times n) + k\rho(16\,(m \times n) + 5\,(m \times n)) \tag{4.14}$$

donde ρ está definido por :

$$\rho = 1 - \frac{\eta}{m \times n} \tag{4.15}$$

donde η es el número de píxeles que no presentaron cambios significativos y k es el número de iteraciones. El número de iteraciones varía según distintos trabajos y van desde 3 iteraciones hasta 500 iteraciones según [MZC+05]. En realidad cuanto mayor sea el número de iteraciones el error disminuirá. Sin embargo, en [HS81] se evaluaron los efectos en el resultado del flujo óptico con 4, 16, 32 y 64 iteraciones. En dicho trabajo se concluyo que después de 32 iteraciones existía muy poca reducción del error y que en otros casos (con otras imágenes) después de 16 iteraciones se presentaba el mismo hecho y que todo dependía del tipo de secuencia de imágenes utilizadas. Lo que si se deja claro en el trabajo es que en las primeras iteraciones el error se reduce de forma considerable y conforme va aumentando el número de iteraciones la reducción del error es menos significativo y en algunas veces es casi nulo.

Una vez analizado el procesado guiado por cambios es necesario contar con un conjunto o secuencias de imágenes para poder iniciar el procesado del flujo óptico guiado por cambio. Algunas de las imágenes utilizadas para realizar el cálculo del flujo óptico fueron tomadas, de la Universidad de Western Ontario, Departamento de Ciencias de la Computación[1] y del grupo de investigación de Visión por Computador, de la Universidad de Otago[2].

Programando y llevando a cabo el procesado del flujo óptico (**FO**) y el procesado del flujo óptico guiado por cambios (**FOGC**) se procede a evaluar la factibilidad y el rendimiento de la técnica propuesta en este trabajo.

4.2.2. El procesado del flujo óptico y su procesado guiado por cambios

Existe un banco de imágenes capturadas y utilizadas comúnmente para el cálculo del flujo óptico. Con el fin de evaluar el sistema aquí propuesto sólo se utilizarán algunas de ellas, de hecho serán las más comunes en la bibliografía. Primero se iniciaran con imágenes capturadas de escenas reales las cuales son capturadas en ambientes cerrados y con cierto control de la iluminación, como es el caso de la secuencia *Rubic*. Después se realizan comparaciones con secuencias sintéticas de las cuales se conoce a priori el movimiento y la velocidad de éste, como es el caso de las secuencias *Diverging tree* (*treed*) y *Traslating tree* (*treet*).

De la secuencia de imágenes *Rubic* se seleccionó un par de imágenes consecutivas (4 y 5) del banco de imágenes de la Universidad de Western Ontario. Las dos imágenes fueron procesadas obteniéndose el flujo óptico correspondiente. Además, se realizó también el procesado del flujo óptico guiado por cambios. En cada uno de los dos procesos se realizó con 10 y 100 iteraciones, y un umbral de $t_h = 1$. El par de imágenes consecutivas y su resultados pueden verse en la figura 4.3. El tamaño de las imágenes originales son de 256×240 píxeles.

Es posible observar que las imágenes obtenidas del procesado del FO para 10 y 100 iteraciones se perciben significativas diferencias en cuanto al flujo óptico se refiere respecto a las imágenes obtenidas del procesado del FOGC para 10 y 100 iteraciones. Pero también se puede ver que son localizados correctamente los puntos donde existe el movimiento puntual. Esto es, se puede apreciar que en la parte

[1] ftp.csd.uwo.ca/pub/vision

[2] http://www.cs.otago.ac.nz/research/vision/

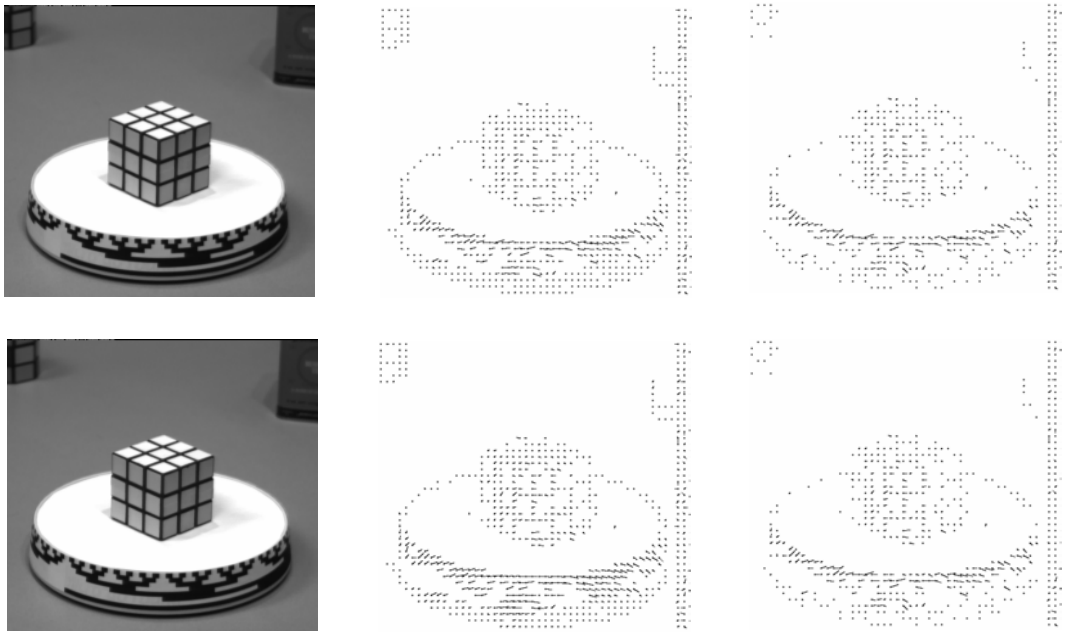


Figura 4.3: *De izquierda a derecha y de arriba hacia abajo: par de imágenes consecutivas originales Rubic(4,5), el cáculo del FO con 10 y 100 iteraciones, y el FOGC con 10 y 100 iteraciones.*

superior izquierda de las imágenes procesadas, área donde no existe movimiento alguno, claramente en el procesado del FOGC no considera movimiento significativo y evita realizar el cálculo de dichos píxeles. Por otro lado identifica claramente el movimiento pero no con la misma densidad de información. Sin embargo, es imposible determinar qué información es la correcta ya que puede darse el caso de que no existiera movimiento pero si cambios en la intensidad.

En el mismo programa se incorporó una rutina para obtener el tiempo de procesado. Los tiempos de cada uno de los procesos se pueden ver en el cuadro 4.1. Del cuadro es posible advertir que el tiempo cuando se ejecutan 10 iteraciones se ve reducido al 66 % del tiempo de procesado del FO. Mientras que cuando se realizan 100 iteraciones el tiempo se ve reducido al 60 % del tiempo necesario para el cálculo del flujo óptico.

Ahora bien, hasta el momento es posible realizar una comparación visual y no formal. Para poder comparar el método de forma más estricta es necesario realizar un análisis del error cuantitativo del flujo óptico. El error se puede expresar como error absoluto o una relación señal a ruido entre el movimiento detectado por el algoritmo para el cálculo del flujo óptico y el flujo óptico real existente. Para conocer

No. Iteraciones	**FO**	**FOGC**
10	141 ms	94 ms
100	1.235 ms	750 ms

Cuadro 4.1: *Tiempos de procesado del flujo óptico (FO) y del flujo óptico guiado por cambios (FOGC) con 10 y 100 iteraciones, para la secuencia de Rubic.*

a priori el flujo óptico real de una escena es necesario obtener imágenes capturadas en ambientes controlados o utilizar secuencias de imágenes sintéticas en las cuales se conocerá el flujo óptico.

En este sentido una estimación del error medio está definido por:

$$E_{prom} = \frac{\sum_0^n E_{rad}}{n} \tag{4.16}$$

donde n es el número total de píxeles y E_{rad} es el error en radianes, que es representado por:

$$E_{rad} = \frac{arc\,cos(\vec{V_C} \cdot \vec{V_E})}{\pi} + \frac{180}{\pi} \tag{4.17}$$

donde $\vec{V_C}$ es la velocidad correcta y $\vec{V_E}$ es la velocidad estimada. Esta medida del error es utilizada en todos los trabajos sobre flujo óptico y convenientemente apropiada para grandes y pequeñas velocidades sin que estas últimas requieran la amplificación inherente en una medida relativa de la diferencia de vectores [BFB94].

Como se mencionó anteriormente, para poder comparar el método y conocer el error es necesario conocer a priori el flujo óptico real. Por esta razón fueron utilizadas dos secuencias de imágenes sintéticas, la *Diverging tree* (*treed*) y *Traslating tree* (*treet*), tomadas del mismo banco de imágenes. El par de imágenes utilizadas fue la 20 y 21, de ambas secuencias de imágenes. Se realizó el procesado del FO y del FOGC, obteniendo los tiempos de procesado y el error medio como se muestra en los cuadros 4.2 y 4.3, para las secuencias *treed* y *treet* respectivamente. En cada uno de los dos procesos se realizaron 10 y 100 iteraciones.

En el cuadro 4.2 se puede apreciar que para 10 iteraciones y un umbral $t_h = 1$ se logra una reducción de tiempo del 14 % cuando se realiza el procesado FOGC, pero el error se ve incrementado en 4,45. Mientras que para 100 iteraciones con un $t_h = 1$ se logra una reducción de tiempo del 13,81 % pero con un aumento del error de 5,33. Para 100 iteraciones con un $t_h = 2$ se reduce un 30 % el tiempo de procesado pero con un aumento del error mucho mayor, de 10,18. La misma tendencia se presenta

No. Iter.	**FO**	**Error**	**FOGC**	**Error**	t_h
10	5,906 s	13,56	5,078 s	18,01	1
100	59,281 s	11,91	51,093 s	17,24	1
100	59,281 s	11,91	41,422 s	22,09	2
100	59,281 s	11,91	35,625 s	25,13	3

Cuadro 4.2: *Tiempos de procesado y error medio del FO y del FOGC con 10 y 100 iteraciones, para el par de imágenes "treed".*

No. Iter.	**FO**	**Error**	**FOGC**	**Error**	t_h
10	5,922 s	41,04	5,437 s	43,92	1
100	59,390 s	38,55	54,343 s	42,72	1
100	59,390 s	38,55	46,563 s	46,72	2

Cuadro 4.3: *Tiempos de procesado y error medio del FO y del FOGC con 10 y 100 iteraciones, para el par de imágenes "treet".*

para 100 iteraciones y con un $t_h = 3$, una reducción en el tiempo pero un incremento del error promedio. Para el caso de la secuencia *treet* los resultados son mostrados en el cuadro 4.3. Los resultados son, en lo general, similares a los de la secuencia anterior. El error relativo crece 2,88 cuando se realiza el procesado del FPGC y el tiempo se ve reducido un 8,18 %, para 10 iteraciones y con un umbral $t_h = 1$. Mientras que para 100 iteraciones y con un $t_h = 2$ el error relativo crece en 8,17 % y el tiempo se ve reducido un 21,59 %.

El hecho de que exista una reducción del tiempo muy pobre y al mismo tiempo un incremento del error medio cuando se realizan 100 iteraciones con umbrales mayores a uno ($t_h > 1$) es debido al tipo de imágenes utilizadas. En las secuencias sintéticas no existen fenómenos como el ruido generado por el sensor o cambios en la iluminación. Son imágenes que en ningún momento presentan condiciones reales de iluminación como distorsiones o efectos del medio ambiente que de una u otra forma afectarían a las escenas reales.

Una secuencia de imágenes sintéticas con un fondo que pudiese generar un efecto de inconsistencia es considerada. La secuencia de imágenes utilizada es la *sphere*. En particular el par de imágenes 1 y 2, del banco de imágenes del grupo de investigación de visión por computadora, de la Universidad de Otago. Las imágenes fueron procesadas obteniéndose el FO correspondiente y el FOGC, con 10 y 100 iteraciones. El umbral utilizado fue de $t_h = 1$. El par de imágenes son de tamaño 200×200 píxeles

y el resultado puede verse en la figura 4.4.

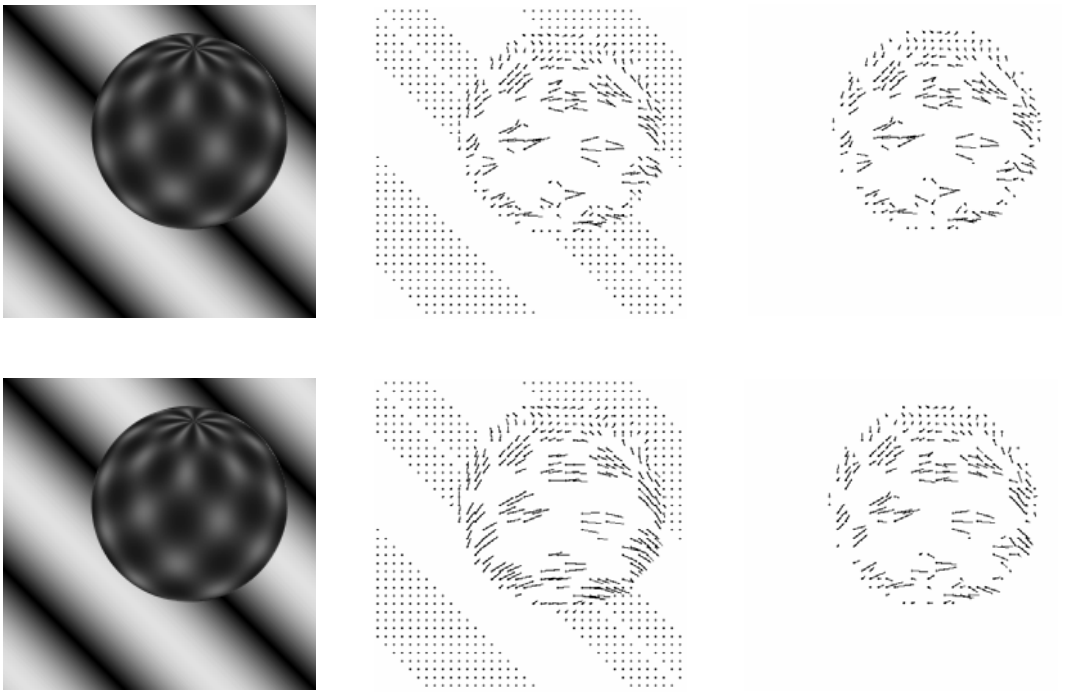


Figura 4.4: *De izquierda a derecha y de arriba hacia abajo: par de imágenes consecutivas originales sphere(1,2), par de imágenes resultantes del cálculo del FO con 10 y 100 iteraciones, y par de imágenes resultantes del FOGC con 10 y 100 iteraciones, con* $t_h = 1$.

Es posible observar que en las imágenes obtenidas durante el procesado del FOGC se elimina ruido o información indebida, provocada por pequeñas variaciones en el fondo de la imagen. Ese hecho no sucede cuando se calcula únicamente el flujo óptico. También se observa que en el caso del procesado del FO se aprecian vectores de flujo óptico bien definidos y con una densidad del flujo un poco mayor. Pero los beneficios que se obtienen en el procesado del FO, en cuanto a una pequeña mejora de la densidad del flujo óptico, se ve oscurecida por el tiempo necesario para el procesado. Para 100 iteraciones el procesado del FOGC se ve disminuido al 29,35 % del tiempo que requirió para realizar el procesado del FO con el algoritmo original, lo que significa una reducción del 70,65 %, como se muestra en el cuadro 4.4.

Además, para esta secuencia particular (*sphere*) el tiempo de procesado del flujo óptico se ve reducido a menos de una tercera parte cuando es utilizada la técnica del procesado guiado por cambios. Además, la calidad del flujo óptico obtenido cuando se realizan sólo 10 iteraciones es prácticamente la misma e incluso se logra eliminar

No. Iteraciones	FO	FOGC
10	94 ms	31 ms
100	1.063 ms	312 ms

Cuadro 4.4: *Tiempos de procesado del FO y del FOGC con 10 y 100 iteraciones y un* $t_h = 1$.

ruido del fondo de la escena. Con esto se puede concluir que la propuesta de este trabajo es viable en cuanto a calidad o eficiencia del sistema. Y abre la puerta para llevar a cabo la implementación del sistema y lograr procesar la información en tiempo real.

Es importante conocer que el número de iteraciones que se pueden aplicar para un procesado en tiempo real difícilmente alcanzarán las 100 iteraciones. Cobos y Monasterio [CM01] únicamente realizaron 3 iteraciones. En [MZC+05] realizaban una iteración por par de imágenes pero el resultado del flujo óptico lo reutilizaban para la iteración del nuevo par de imágenes, esto se repetía durante 64 pares de imágenes. En este trabajo se propone hacer entre 10 y 20 iteraciones para el procesado del FOGC. La razón es debido a que los resultados obtenidos se aprecia que cuando se realiza el procesado haciendo 10 iteraciones el error crece marginalmente y el tiempo se reduce, dependiendo de la secuencia de la imagen estudiada, que va del orden del 14 al 67 % del tiempo total utilizado por el algoritmo original para el cálculo del flujo óptico.

4.3. Consideraciones para el diseño

Una vez hecho el análisis y el procesado del FOGC es posible concluir que la detección de los píxeles que han presentado cambios significativos, por arriba de un umbral establecido, se verán reducidos los datos a procesar. Por lo tanto de manera directa también se verá reducido el tiempo de procesado.

Para poder implementar la técnica aquí propuesta y garantizar unas buenas prestaciones es necesario tomar en cuenta varias consideraciones.

La primera de ellas es realizar la captura del par de imágenes consecutivas, la de referencia (Im_{t-1}) y la actual (Im_t), con una frecuencia de muestreo alta. Esto es porque cuanto menor sea el tiempo, entre la captura de una imagen y la siguiente, menos cantidad de cambios existirán. Así mismo, se puede dar el caso de que si

dos imágenes consecutivas son capturadas con un pequeño intervalo de tiempo de diferencia y no existe cambio alguno entre los píxeles de dichas imágenes, entonces no se realizará procesado alguno. Sólo hasta que una nueva imagen sea comparada con la imagen de referencia y se detecte cambio alguno, se realizará el procesado de la información.

La segunda consideración es tener un módulo para detectar sólo los píxeles que han presentado cambios significativos, respecto a un umbral. El umbral deberá adaptarse al medio o bien a los cambios de intensidades para así llevar a cabo una mejor discriminación de la información relevante y la no relevante.

Una tercera consideración aparece cuando el par de imágenes consecutivas (Im_{t-1} e Im_t) difieren totalmente una de otra, un caso que no es posible en la vida real, ya que cualquier móvil u observador nunca captaría dos escenas totalmente diferentes y menos cuando se utilice una alta frecuencia de muestreo. Sin embargo, en dicho caso es lógico suponer que el tiempo de procesado del FOGC debería ser mayor al tiempo de procesado del FO utilizando el algoritmo original. Esto sería totalmente cierto si el procesado de la información fuera completamente secuencial o se esté utilizando la arquitectura convencional. Este inconveniente puede ser descartado si se implementa el módulo de detección de los píxeles relevantes simultáneamente con la primera etapa del procesado del flujo óptico. Así, en caso de que los píxeles hayan cambiado se continuaría con el procesado de esa información. En el otro caso, de que los píxeles no cambiaron, por lo que no resultan relevantes para el sistema serían descartados automáticamente y por consiguiente no serían procesados en las etapas posteriores.

Se puede decir que en todos los casos se reduciría el número de accesos a memoria, lo que conduce a una reducción en el tiempo de procesamiento del sistema.

Capítulo 5

Diseño hardware de la arquitectura

5.1. Introducción

En el capítulo 3 se estudiaron varias arquitecturas hardware para el procesado del flujo óptico. En particular, en la sección 3.3.2 fue donde se abordaron dichas arquitecturas hardware. Allí se detallaron los recursos hardware necesarios para llevar a cabo el cálculo del flujo óptico, dejándose claro que las arquitecturas requieren una gran cantidad de memoria local para llevar a cabo el procesado de la información. En el capítulo 4 se expuso la técnica para el procesado del flujo óptico guiado por cambios, además de las consideraciones necesarias para hacer eficiente la arquitectura propuesta.

En este capítulo se realiza el diseño hardware de la arquitectura propuesta. Se presenta el diseño de cada una de las etapas que conforman el sistema, para el procesado del flujo óptico guiado por cambios, mostrado en la figura 5.1. La estrategia de diseño es del tipo *top-down* (de arriba hacia abajo). La razón de utilizar este tipo de metodología de diseño es que permite realizar una implementación partiendo de un nivel de abstracción alto e ir hacia abajo incrementando el nivel de detalle según sea necesario. La ventaja de esta metodología se podrá apreciar más adelante, durante el diseño de los módulos.

El punto de partida del diseño será el diagrama a bloques mostrado en la figura 5.1. Por otro lado, considerando la metodología de diseño empleada es posible que algún módulo pueda ser incluido en otro módulo o bloque. Esto significa que la

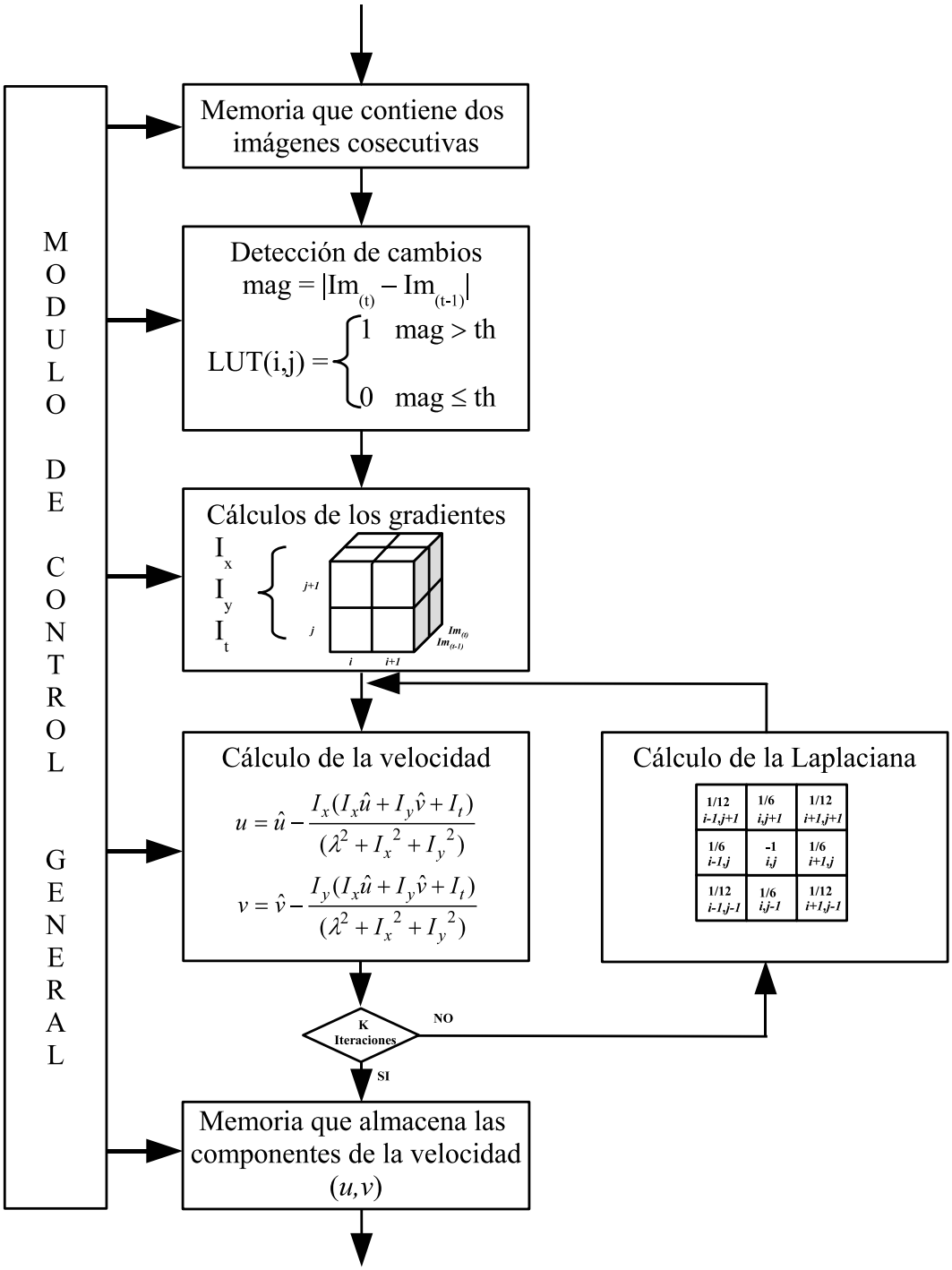


Figura 5.1: *Diagrama general a bloques del sistema de procesado de flujo óptico guiado por cambios.*

metodología de diseño permite tener una visión global de las funciones que se realizan en cada una de las etapas en que se ha dividido el sistema.

Como una primera evaluación, el diseño del sistema fue dividido en 5 módulos o etapas siguiendo el algoritmo utilizado. La división natural fue el resultado de haber realizado la implementación por software del algoritmo propuesto por Horn y Schunck para el cálculo del flujo óptico. Adicionalmente se incorporaron dos bloques uno para la detección de cambios y otro que sería el módulo de control del sistema. Durante el diseño de cada uno de los módulos se cuidó el poder llevar a cabo una ejecución en tiempo real y reducir el costo hardware necesario para la implementación física. Todo eso sin descuidar la calidad del resultado obtenido, respecto a la calidad de trabajos presentados en la literatura. Siguiendo estos criterios cada uno de los módulos fue diseñado y evaluado de forma independiente. Una vez evaluados se procedió a relacionarlos para conformar el diseño completo y realizar una evaluación global del sistema. Los módulos en que fue dividido el sistema son:

1. **Módulo de detección de cambios.** Este módulo realiza la detección de los cambios significativos entre las dos imágenes consecutivas. El resultado esperado es una tabla LUT que contendrá la posición de los píxeles que mostraron cambios significativos, respecto a un umbral establecido.
2. **Módulo del gradiente.** Aquí se calculan las derivadas espacio-temporales. En principio serán procesados sólo los píxeles que han presentado cambios significativos proporcionados por el módulo de detección de cambios. El resultado serán tres valores de 9 bits en complemento a dos que representan los gradientes espacio-temporal (I_x, I_y e I_t), pero aunque son calculados todos los gradientes de todos los píxeles de las imágenes sólo serán procesadas las componentes de los gradientes que pertenecen a los píxeles que presentaron cambios significativos.
3. **Módulo de velocidad.** Esta etapa consiste en calcular los componentes de la velocidad (u, v) de los píxeles que presentaron cambios significativos. Es la etapa que más recursos hardware utiliza pues realiza operaciones con un alto coste de diseño como son las divisiones. Por otro lado aquí es donde se presenta el cuello de botella del sistema, reduciendo la frecuencia de cálculo.
4. **Módulo Laplaciana.** Esta parte del sistema realiza el suavizado de la imagen, restricción necesaria para obtener una solución a la ecuación del flujo óptico. El suavizado consiste en aplicar una máscara de 3×3 píxeles, sobre el campo del flujo óptico en sus dos componentes de la velocidad. Los valores de la máscara tienen diferente peso, según sea la posición del píxel en la máscara.

5. **Módulo de control.** Este módulo se encargará de sincronizar todos los módulos del sistema, de acceder a las memorias tanto de cada módulo como a la memoria principal del sistema y de indicar qué módulo debe parar o iniciar el proceso.

Es importante resaltar que en cada uno de los módulos y en el diseño total del sistema, se deberá procesar la información en tiempo real. Dicho de otra forma, el sistema deberá tener la capacidad de procesar al menos 24 fps. Para conseguir esta capacidad de procesado se propone trabajar a una frecuencia de reloj de 33 MHz y con imágenes de dimensiones no muy grandes, según trabajos propuestos en la literatura estos tamaños son de 256×256 píxeles [MZC+05]. Aquí se propone utilizar imágenes de las misma dimensiones, 256×256 píxeles.

5.2. Módulo de detección de cambios

Este módulo es el más sencillo del sistema, por lo que utiliza pocos recursos hardware y además se ejecuta en poco tiempo. La función que realiza este módulo consiste básicamente en realizar una resta entre dos imágenes consecutivas e identificar los píxeles que cambiaron entre el par de imágenes consecutivas. Las diferencias serían todos aquellos cambios significativos, que se encuentran por arriba de un umbral específico previamente establecido, entre dos píxeles del par de imágenes consecutivas. Entonces este proceso se realiza una vez que se han capturado y almacenado en la memoria local del sistema el par de imágenes consecutivas. De la figura 5.1 se puede observar que primero se capturan las dos imágenes consecutivas y éstas son almacenadas en la memoria local del sistema. Posteriormente se lleva a cabo la detección de los cambios entre las dos imágenes para después continuar con la siguiente etapa del sistema, que en este caso sería el cálculo de los gradientes espacio-temporales.

Para llevar a cabo el proceso de la detección de los cambios se utiliza un circuito restador por cada par de píxeles, de las dos imágenes consecutivas. El inconveniente para llevar a cabo una resta simultánea de las dos imágenes en paralelo es la gran cantidad de memoria necesaria dentro de la FPGA para su procesado. Suponiendo que se desea procesar imágenes de 256×256 a B/N, entonces serían necesarios $2 \times 256 \times 256 \times 8 + 256 \times 256 \times 9 = 1.638.400$ bits de memoria y los circuitos restadores necesarios. Todos eso resulta de la necesidad de almacenar las dos imágenes consecutivas con píxeles de 8 bits, manteniéndolas intactas para el futuro procesado, y una tercera imagen para almacenar la *imagen diferencia*, que puede tener valores

negativos (9 bits). La cantidad de memoria resultaría muy costosa para una FPGA e imposible muchas veces en caso de querer implementar otras operaciones dentro de la misma FPGA. Además se deben considerar los componentes necesarios para efectuar los 256×256 circuitos restadores.

Otra alternativa, totalmente opuesta, sería realizar la operación de dos píxeles en dos píxeles. Su desventaja, claramente visible, es que se estaría realizando un procesado similar al realizado en un computador personal, que sería un procesamiento secuencial. En este caso se realizan los accesos a memoria de los píxeles conforme se vayan necesitando. Este hecho agrega de manera indirecta un aumento de tiempo o bien no se aprovecharía el beneficio de los accesos a memoria en ráfagas. Los accesos en ráfagas permiten disminuir el número de ciclos necesarios para el acceso a memoria pero la restricción es que los datos a acceder deberán ser direcciones consecutivas, a diferencia del acceso a píxeles independientes. Además al realizarse un procesado secuencial sería un mecanismo poco acertado debido a la capacidad y el beneficio hardware que proporcionan las arquitecturas reconfigurables. Por estas razones se propone utilizar fracciones parciales de las imágenes, procesarlas y acceder por un nuevo segmento de datos de las imágenes en cuestión, en este caso sería de un nuevo renglón de cada imagen.

Por otro lado, también se requiere un circuito que realice la comparación de la diferencia obtenida (*magnitud*) con el valor de un umbral establecido. Una vez que sean identificados los píxeles, que muestran cambios significativos en el tiempo, entonces su dirección será almacenada en una tabla (LUT) que servirá para localizar esos píxeles específicos. La tabla puede ser almacenada en la misma memoria local, donde se encuentran las dos imágenes consecutivas, pero también puede ser almacenada en la memoria de la FPGA puesto que comúnmente contienen un área de memoria predefinida.

La idea de almacenar la LUT dentro de la FPGA pretende agilizar el procesado de la información. De hecho se puede decir que la información contenida en la LUT es la base del sistema debido a que será indispensable conocer los píxeles que cambiaron. Este hecho, el conocer la localización de los píxeles que cambiaron, es indispensable durante las etapas siguientes del procesado de la información, como el cálculo de los gradientes, de la velocidad y de la Laplaciana. Por eso, el contenido de la tabla es una información muy utilizada y esencial para el **procesado guiado por cambios**.

Si el tamaño de la imagen utilizada es de 256×256, entonces el tamaño de la palabra requerida para localizar un píxel de la imagen es de 16 bits. Esto hace que

se requieran al menos 1,05 Mbits de memoria, una cantidad considerable y más aún si es necesario implementar otros módulos que requieran más memoria dentro de la misma FPGA. Por esa razón se propone realizar una LUT que contendrá sólo un bit por píxel. Si el valor del bit es cero significaría que el píxel, de la imagen de referencia y actual, no presentó cambio alguno. Si el valor del bit es 1 entonces indica que dicha posición del píxel sí presentó cambio, respecto a un umbral previamente establecido. De esta manera al utilizarse un bit por píxel se requiere una memoria de sólo 65.536 bits.

La desventaja de contar con un bit por par de píxeles hace que sólo se conozca si el píxel deberá ser procesado o no, pero nunca la posición dentro de las imágenes. Este hecho hace necesaria una lógica adicional para conocer la localización de los píxeles que deberán ser procesados. La misma lógica leerá la LUT y deberá saber la posición correspondiente en la memoria principal del bit leído en la tabla. Entonces es necesario un circuito generador de direcciones en función de la posición del píxel dentro de la tabla (LUT). Una posible solución es utilizar un contador de módulo de 16 bits, para que pueda acceder a toda la LUT y con un off-set se genera el valor de la dirección, esto por cada bit leído. Así, cuando inicie la cuenta el circuito contador se realizará una lectura secuencial del contenido de la tabla. Pero si además se considera un ancho de palabra de 8 bits en la LUT, entonces se podrán leer o conocer hasta 8 direcciones en un ciclo de reloj. A su vez el módulo del contador se reduciría a 13 bits. En caso de que ningún bit, de los 8 que serán leídos en cada ciclo de reloj, haya sufrido algún cambio significativo o al haber concluido de leer los 8 bits se procede a leer la la siguiente palabra de la LUT. Los valores leídos serán almacenados en una nueva LUT2 la cual será leída en una etapa siguiente del proceso.

Si continuamos con la secuencia del diagrama de flujo mostrado en la figura 5.1, se tiene que una vez que se cuenta con la información de los píxeles que han cambiado, localizados en la LUT, es posible iniciar con la siguiente etapa del proceso. Para esto es necesario primero leer la LUT y localizar los píxeles en la memoria principal local. El número de ciclos de reloj necesarios para leer la LUT es de 8.192, independientemente del número de píxeles que presentaron cambios y es posible leer hasta 8 valores por ciclo de reloj. Esto se traduce a localizar 16 píxeles en la memoria local, siempre y cuando los 8 píxeles hayan presentado cambios significativos. Son 16 accesos a memoria por que serían a 8 píxeles de cada una de las imágenes. Entonces se requieren un total de 131.072 ciclos de reloj para acceder de forma independiente a cada una de las imágenes. Todo esto es en el peor de los casos, que es cuando

las imágenes de referencia y actual son totalmente diferentes, aunque es muy poco probable y casi imposible de que se presente este hecho.

Desde un punto de vista global del sistema, se puede apreciar que este módulo presenta algunas características y operaciones semejantes a las realizadas en el módulo siguiente, el módulo del gradiente. En ambos módulos se utilizan datos de las dos imágenes consecutivas, acceden por los datos a la memoria principal local y también realizan una substracción entre la posición de un píxel de la imagen de referencia y la imagen actual. El módulo del gradiente ejecuta la operación para el cálculo del gradiente I_t la cual es similar a la realizada en la etapa de detección de cambios. Por esta razón es posible que la etapa de detección por cambios se implementen en paralelo a la etapa del calculo del gradiente e incluso utilizando parte de la arquitectura utilizada en el cálculo del gradiente. Con esta estrategia se busca reducir el hardware necesario para la implementación de ambos módulos y reducir el número de accesos necesarios a la memoria local.

5.3. Módulo del gradiente

Durante el diseño del módulo anterior se realizó una primera aproximación del módulo de detección de cambios y la información que proporcionaría al módulo del gradiente. El diseño siguió una implementación directa. Dicho en otras palabras, la fase de diseño siguió el diagrama de bloques de la figura 5.1. Eso significa que primero se realiza la identificación de los píxeles relevantes creando la tabla LUT y posteriormente se lleva a cabo el cálculo de los gradientes espacio-temporal. Sin embargo, existen dos inconvenientes muy importantes al no haber considerado la estrategia de diseño integral o con una visión más global de ambos módulos.

El primer inconveniente es el incremento del número de accesos a memoria, aproximadamente $3(m \times n)$ veces más, donde $(m \times n)$ es el tamaño de la imagen, que si sólo se hubiera realizado el cálculo espacio-temporal, debido a una implementación secuencial. Lo antes dicho es sólo si se considera que las dos imágenes consecutivas fueran completamente diferentes, en otro caso el incremento sería relativo en función del número de píxeles que hayan presentado cambios significativos.

El segundo inconveniente aparece cuando se realiza el cálculo de la derivada espacio-temporal de un píxel específico. El píxel a procesar será aquel que haya presentado un cambio significativo, que será proporcionado por el módulo de detección de cambios. Sin embargo, para realizar el cálculo del gradiente de ese píxel específico

es necesario conocer el valor de los píxeles vecinos. Esto a su vez requiere acceder a la memoria nuevamente un número determinado de veces igual al número de píxeles vecinos que sean necesarios para llevar a cabo el cálculo del gradiente. En este caso se requieren 8 píxeles para el procesado del gradiente incluyendo el píxel indicado por el módulo de detección de cambios. Entonces el módulo de detección de cambios sólo tendrá localizados los píxeles que presentaron cambios significativos y el módulo del gradiente debe generar las direcciones adicionales para acceder a las localidades vecinas necesarias para llevar a cabo el cálculo del gradiente.

Ambos inconvenientes pueden ser superados si se realiza el diseño contemplando una visión integral del sistema. Entonces si se consideran las necesidades de cada uno de los módulos es posible mejorar el rendimiento de la arquitectura hardware y realizar un diseño óptimo. De esta manera es posible visualizar varias similitudes entre la etapa de detección de cambios y la etapa para el cálculo del gradiente. Algunas de las coincidencias entre estas dos etapas son:

1. El módulo de detección por cambios debe acceder a la memoria local para leer todos los píxeles de ambas imágenes. El módulo del gradiente requiere información de los píxeles que presentaron cambios así como de sus píxeles vecinos, aunque estos últimos no hayan presentado cambio alguno. Esto significa que se requiere información adicional y por otro lado los datos que se requieren ya habían sido leídos de la memoria. Lo que se traduciría en una doble o hasta cuádruple lectura de un mismo píxel aunque no haya presentado cambio alguno.
2. Los datos de entrada a procesar para ambos módulos son los mismos.
3. Existe una operación común, y con los mismos datos de entrada, en ambos módulos. La operación es una sustracción.
4. Cuantos más píxeles se procesen en paralelo más rápido será ejecutado cada uno de los módulos.
5. El módulo del gradiente requiere necesariamente 8 píxeles, 4 de cada imagen. Los mismos píxeles son utilizados en el módulo de detección de cambios y podría procesar 4 pares de píxeles en paralelo.

Entonces sí se puede calcular simultáneamente el gradiente espacio-temporal y el módulo de detección de cambios, es factible concluir que se optimizará la arquitectura y se mejorarán los tiempos de procesamiento. La mejora en los tiempos de procesado es un resultado implícito al realizar un diseño en paralelo. Este tipo de diseño siempre producirá una reducción del tiempo de procesado. Adicionalmente

el leer los datos de memoria y utilizarlos para ambas etapas del sistema se llevará a cabo una reducción del número de accesos a memoria. Desde el punto de vista hardware esta estrategia optimiza la arquitectura utilizada, ya que en ambos módulos existen operaciones comunes y entonces el mismo hardware es utilizado en ambos procesos.

Por otro lado, si se analiza el proceso y la secuencia de los píxeles necesarios para llevar a cabo el cálculo del gradiente espacio-temporal, es posible hacer más eficiente la arquitectura. Para calcular el gradiente espacio-temporal de un píxel $P(i, j)$ se requieren 8 píxeles, 4 píxeles por imagen, como se muestra en la figura 5.2.

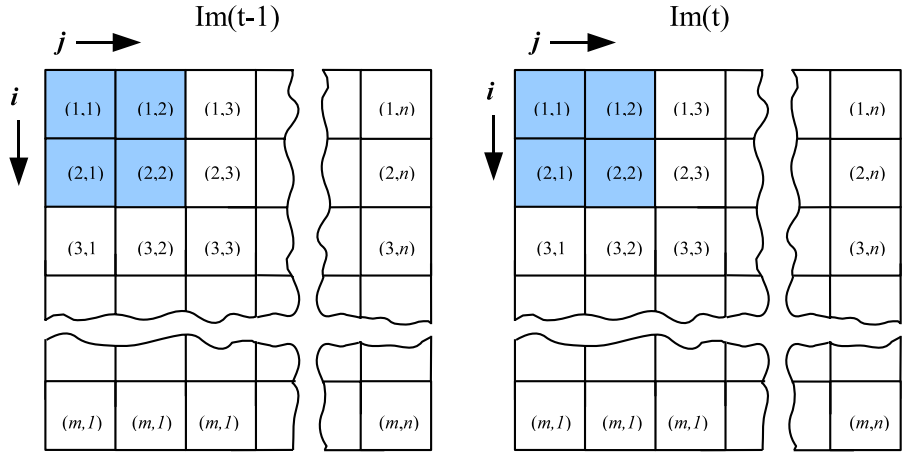


Figura 5.2: *Píxeles necesarios, del par de imágenes consecutivas para el cálculo del gradiente espacio-temporal.*

Por ejemplo, para calcular el gradiente $I_x(i, j)$ de la fila i y la columna j, del píxel $P(2, 2)$, la secuencia de lectura de memoria sería la siguiente:

$P_{in}(1, 1, t-1)$ – $P_{in}(1, 2, t-1)$ – $P_{in}(2, 1, t-1)$ – $P_{in}(2, 2, t-1)$ – $P_{in}(1, 1, t)$ – $P_{in}(1, 2, t)$ – $P_{in}(2, 1, t)$ – $P_{in}(2, 2, t)$

A partir de éstos 8 valores, se obtiene el primer valor de salida. Para obtener el siguiente gradiente de salida $I_x(i, j+1)$ se requerirían 8 nuevos valores, de los cuales 4 de ellos coinciden con los anteriores. Esto se puede observar en la secuencia para obtener el siguiente gradiente de salida:

$P_{in}(1, 2, t-1)$ – $P_{in}(1, 3, t-1)$ – $P_{in}(2, 2, t-1)$ – $P_{in}(2, 3, t-1)$ – $P_{in}(1, 2, t)$ – $P_{in}(1, 3, t)$ – $P_{in}(2, 2, t)$ – $P_{in}(2, 3, t)$

Todo esto da una idea de que es posible reducir el número de accesos a memoria, ya sea procesando una mayor cantidad de píxeles en paralelo o utilizando una memoria para almacenar temporalmente el valor de los píxeles que serán utilizados posteriormente. Esta última opción puede ser extendida tanto para el cálculo del siguiente píxel como para el píxel que se encuentra un renglón abajo, $(i+1, j)$ píxeles después. De este modo si este principio se extrapola, no sólo considerando dos píxeles adicionales, considerando dos renglones completos de cada una de las imágenes es posible asegurar que se reducirá considerablemente el número de accesos a memoria. Por otro lado también se asegura el flujo de datos para tener un procesado constante. Así surge la propuesta mostrada en el diagrama de la figura 5.3.

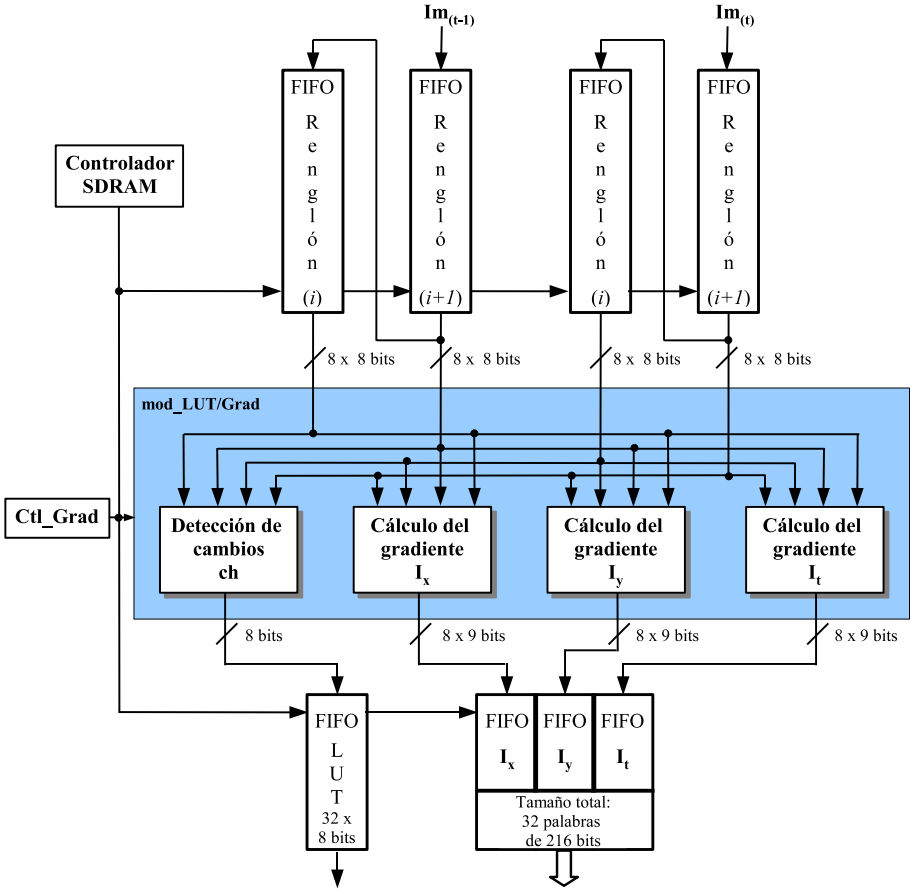


Figura 5.3: *Diagrama de interconexión entre las memorias FIFO's de entrada y de salida con el módulo del gradiente (mod_LUT/Grad).*

Cada memoria FIFO de entrada podrá almacenar n píxeles y tendrá un ancho de bus de 8 píxeles (64 bits), donde n es el número de píxeles de cada renglón de una

imagen de tamaño $(m \times n)$ píxeles. En este caso se utilizan imágenes de 256×256. El módulo `LUT/Grad` está compuesto por 4 bloques, uno realiza la detección de cambios y los otros tres calculan los gradientes espacio-temporales I_x, I_y e I_t. La función que realizan cada uno de estos tres bloques es idéntica, y consiste básicamente en sumas, restas y dos desplazamientos a la derecha (para ejecutar una división por 4), la diferencia radica en los datos que ingresan. La división es una operación necesaria para realizar el cálculo de los gradientes. La salida de cada bloque que calcula un gradiente es de 9 bits, debido a que puede dar valores negativos. En el caso del bloque de detección de cambios (***ch***), se tiene una salida de 8 bits y los datos son almacenados en una memoria FIFO llamada `LUT`. La finalidad de almacenar un bit por píxel es para reducir la cantidad de memoria necesaria y poder almacenar la LUT dentro de la misma FPGA. El bit almacenado puede tomar el valor de 0 y 1, será en función de que se haya detectado un cambio entre los píxeles de una posición(i, j) de las dos imágenes consecutivas. Si los píxeles presentaron un cambio significativo, por arriba de un umbral establecido, entonces se almacena un 1 lógico en la LUT, en otro caso se almacena un 0 lógico.

El módulo `LUT/Grad` presenta una latencia de un ciclo de reloj y procesa 8 píxeles simultáneamente. El flujo de datos es controlado por el módulo `Ctl_Grad`, el cual es un submódulo de control que se verá detalladamente en la sección 5.6. El módulo controla la lectura de la SDRAM, inicialmente de 4 renglones, y la escritura en la memoria FIFO de entrada. Una vez que `Ctl_Grad` detecta que la memoria FIFO de entrada está llena, le indica al módulo `LUT/Grad` que procese 32 palabras. Un ciclo después de que inicia el procesado se obtienen resultados que serán leídos y escritos en las FIFO's de salida. Todo esto es controlado por el módulo `Ctl_Grad`. Simultáneamente durante el procesado de la información el contenido de los renglones *(i+1)* de Im_{t+1} e Im_t, de la FIFO de entrada, se desplazan a los renglones *(i)* para cada caso, como se aprecia en al figura 5.3. De tal forma que una vez procesadas las 32 palabras los renglones *(i+1)*, de Im_{t+1} e Im_t almacenados en la FIFO de entrada, estarán vacíos. Por lo tanto el módulo `Ctl_Grad` realizará una petición de lectura a la memoria SDRAM de dos nuevos renglones, a través del controlador de la SDRAM. Los datos leídos, un renglón de cada imagen, son almacenados en la memoria FIFO de entrada para preparar un nuevo ciclo de procesado del módulo `LUT/Grad`.

Los bloques que calculan cada uno de los gradientes I_x, I_y e I_t proporcionan tres salidas de 72 bits cada una, correspondiente a 8 píxeles de salida en paralelo. Cada píxel de salida es de 9 bits (signo-complemento a 2). Las salidas son almacenadas

en una memoria FIFO de 32 palabras con un ancho de 216 bits. En la figura 5.4 se representa la memoria FIFO como 3 FIFO's independientes pero es sólo para dejar más claro cuales son las componentes obtenidas y para qué píxeles. La `LUT`, que almacena los bits de cambios, tiene un tamaño de (32×8) bits, esto es 32 palabras con un ancho de 8 bits cada una de ellas.

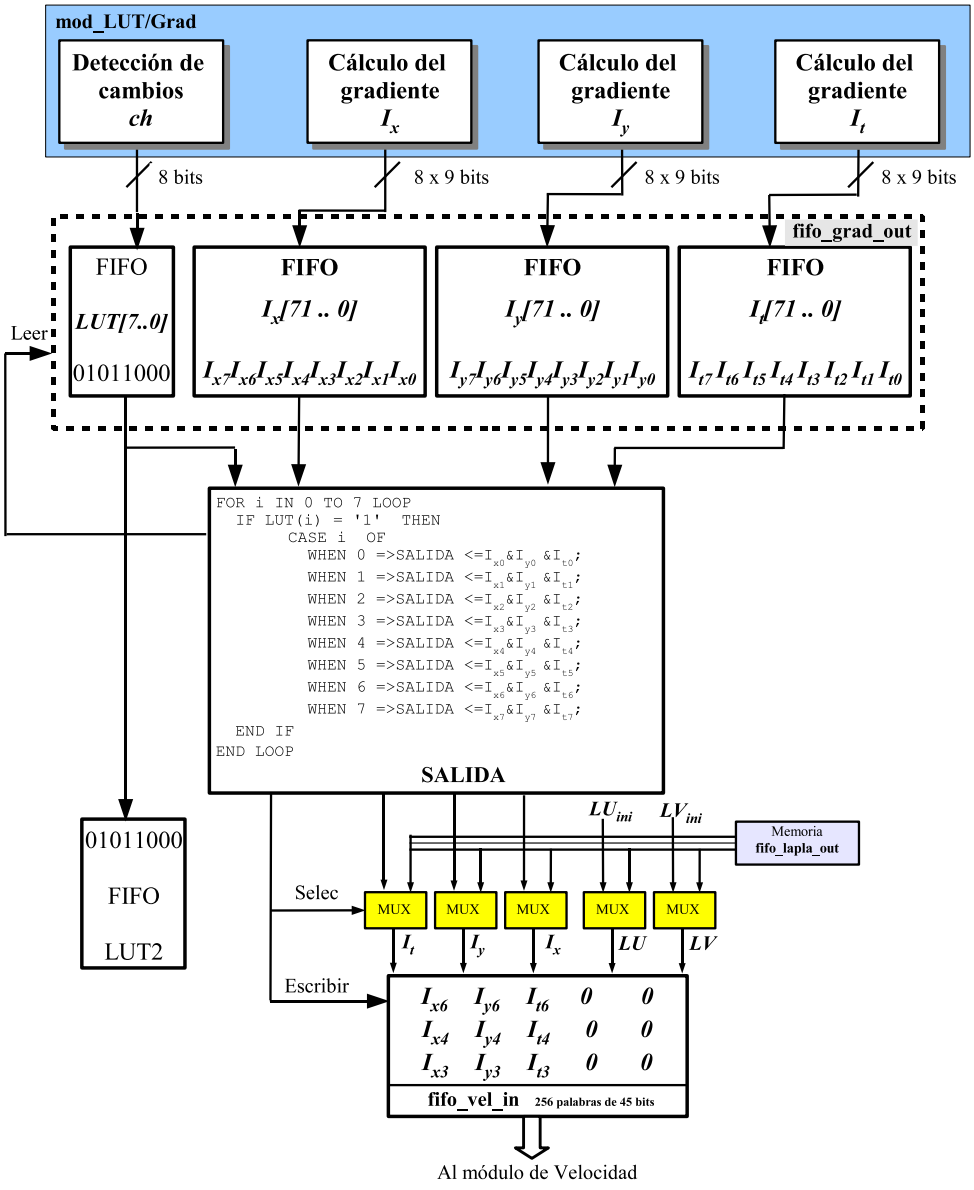


Figura 5.4: *Gestión conjunta de la LUT y la fifo_grad_out, para seleccionar los valores de los gradientes que sí cambiaron.*

El circuito que controla el acceso a la `LUT` para leer los datos e iniciar con la selección de las componentes necesarias para el procesado es el módulo `Ctl_grad_lap`.

Este proceso se realiza cuando se cuenta con la primera palabra en la tabla `LUT`, lo que significa que se ejecuta en paralelo con la ejecución del módulo `LUT/Grad`. Parte del circuito descrito en VHDL se muestra en la figura 5.4, donde se presentan también algunos bloques que intervienen en el proceso. De hecho, en la figura se muestra un ejemplo de la lectura de un renglón de la `LUT`, la lectura y escritura de los gradientes correspondientes a los píxeles que cambiaron. Al leer la `LUT` se obtiene la palabra [01011000] de tal manera que los píxeles que cambiaron fueron el tres, cuatro y seis de ocho [7..0]. Así sólo se seleccionan las componentes de los gradientes que comprenden a esas posiciones y son escritas en una nueva memoria llamada `fifo_vel_in`, como se puede apreciar en la figura 5.4. En la misma figura también se puede apreciar que existen dos columnas que corresponden a las componentes de la laplaciana las cuales inicialmente son cero.

El proceso de selección de los gradientes que corresponden a los píxeles que sí presentaron algún cambio significativo, se realiza en 8 ciclos de reloj. Al concluir la lectura de cada una de las palabras de la `LUT` será almacenada en una nueva tabla `LUT2` y leída una nueva palabra de la `LUT`. Si se pretende tener una mayor eficiencia en el sistema es posible utilizar memorias de doble puerto con dos señales de reloj, una para lectura y otra para escritura. De esta manera es posible considerar leer la `LUT` y las FIFO's I_x, I_y e I_t a una frecuencia más alta con la finalidad de mantener un flujo de datos constante.

El módulo `Ctl_grad_lap` realiza adicionalmente otra función: Cuando se realiza la segunda iteración o posteriores debe leer las componentes necesarias para efectuar el cálculo de la velocidad de la memoria `fifo_lapla_out` y escribirlas en la memoria `fifo_vel_in`. Las componentes leídas y escritas son LU, LV, I_x, I_y e I_t.

El código VHDL que describe al módulo `LUT/grad` y al circuito `Ctl_grad_lap` se encuentra en el apéndice A.

5.4. Módulo de velocidad

El módulo de velocidad es la etapa que requiere más recursos debido a que realiza las operaciones más complejas del sistema. Por otro lado este bloque es el cuello de botella del sistema, en referencia a la frecuencia de procesado. Por este motivo se realiza una segmentación del módulo de velocidad en dos etapas. De este modo la función original para el cálculo de cada una de las componentes de la velocidad (u, v) deberá ser dividida en varios términos, como se muestra la ecuación 5.1.

$$(u,v) = \begin{cases} u = \overline{u} - I_x \frac{I_x\overline{u}+I_y\overline{v}+I_t}{\lambda^2+I_x^2+I_y^2} = \overline{u} - I_x\frac{TP}{TD} \\ v = \overline{v} - I_y \frac{I_x\overline{u}+I_y\overline{v}+I_t}{\lambda^2+I_x^2+I_y^2} = \overline{v} - I_y\frac{TP}{TD} \end{cases} \tag{5.1}$$

Las operaciones entre los términos deben ser ejecutadas considerando que los cálculos son hechos en arquitecturas que soportan sólo operaciones aritméticas de números enteros con signo. El motivo de realizar operaciones con números enteros es porque requieren pocos recursos hardware, optimizando el uso de los recursos, y a la vez se incrementa la velocidad de procesado del sistema.

De esta manera, de la ecuación 5.1 se puede observar qué operaciones deberían ser efectuadas antes y cuales después. Se puede ver que los términos *TP* y *TD* son los primeros en calcularse. Es posible ver que dichos términos son iguales en ambas componentes de la velocidad. Por lo tanto se podría pensar que deberían ser calculados como un solo término (TP/TD). Pero no se realiza de esa forma debido a que se utiliza aritmética con números enteros y al realizar primero una división y después un producto es posible perder información. Entonces con la finalidad de reducir el error debido al truncado del término al efectuar la división, se realizarán primero los productos de $(I_x \cdot TP)$ e $(I_y \cdot TP)$ y posteriormente se efectuará la división. Así cada uno de los dos productos serán divididos entre el término *TD*.

Para visualizar de forma general cada uno de los términos y su consecutivo procesado, la ecuación 5.1 puede ser reescrita como:

$$(u,v) = \begin{cases} u = \overline{u} - I_x \frac{I_x\overline{u}+I_y\overline{v}+I_t}{\lambda^2+I_x^2+I_y^2} = \overline{u} - I_x\frac{TP}{TD} = \overline{u} - \frac{TP_x}{TD} = \overline{u} - R_x \\ v = \overline{v} - I_y \frac{I_x\overline{u}+I_y\overline{v}+I_t}{\lambda^2+I_x^2+I_y^2} = \overline{v} - I_y\frac{TP}{TD} = \overline{v} - \frac{TP_y}{TD} = \overline{v} - R_y \end{cases} \tag{5.2}$$

Para el cálculo del término *TD* es necesario conocer sólo los gradientes espaciales (I_x, I_y) y la variable λ. La variable λ es utilizada cuando ambos gradientes son cero, entonces ésta tomará el valor de uno ($\lambda = 1$), en cualquier otro caso tomará el valor de cero ($\lambda = 0$). Con la finalidad de reducir recursos hardware y por la característica particular del término *TD*, que es la suma de dos valores al cuadrado, siempre será el resultado positivo, entonces sólo se ingresará la magnitud de los dos valores (8 bits) y el resultado puede ser escrito en un registro de 17 bits sin signo. Este hecho evita realizar dos productos con términos de 9 bits y a cambio sólo se utilizan términos de 8 bits, como se puede observar en la figura 5.5.

Para el cálculo del término *TP* son necesarios los tres gradientes (I_x, I_y, I_t) y las componentes del promedio de las velocidades $(\overline{u}, \overline{v})$, llamadas Laplacianas. Estas

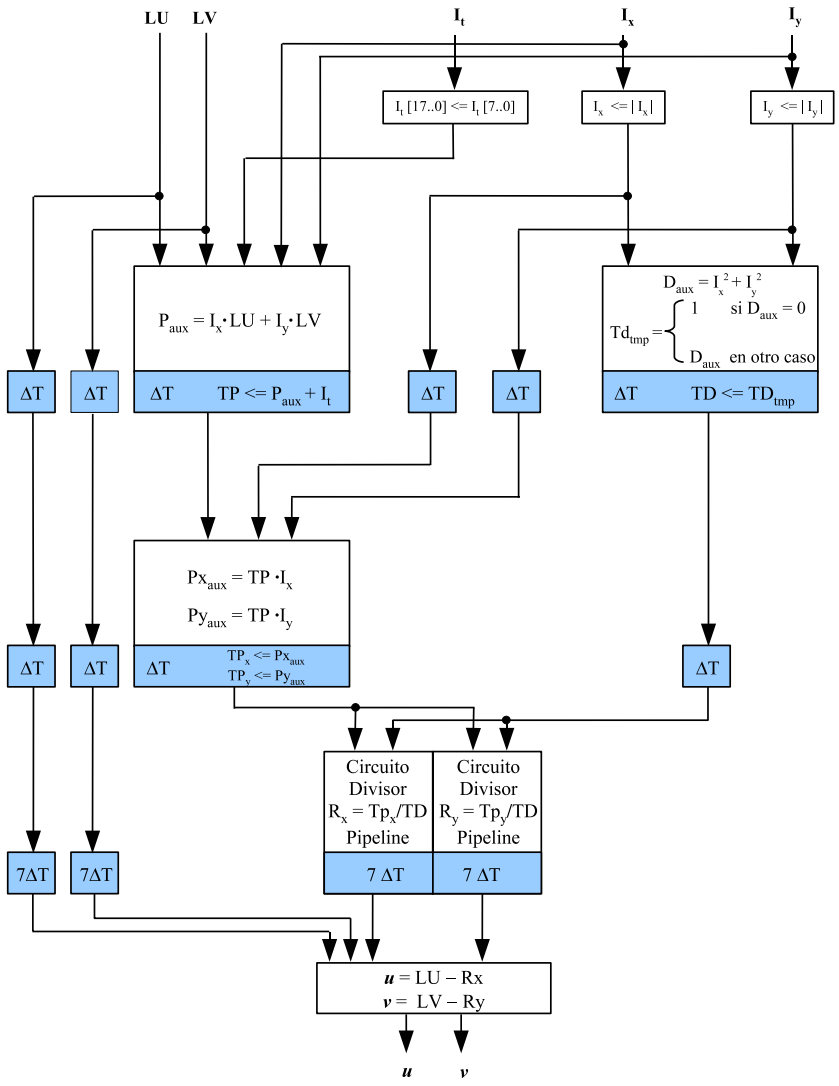


Figura 5.5: *Diagrama esquemático del módulo velocidades.*

últimas componentes confirman la característica iterativa del cálculo del flujo óptico, pues su valor depende del cálculo de la velocidad. Entonces como no existen valores previos, para la primera iteración, los valores iniciales de las componentes $(\overline{u}, \overline{v})$ serán cero.

Hasta este momento está claro que ambos términos (TP y TD) son de un tamaño de 18 y 17 bits respectivamente. Los términos pueden ser calculados en paralelo, lo que significa que no dependen un término del resultado del otro. Además, debido al tipo de operaciones que se realizan es posible ejecutarlas en un ciclo de reloj. Por otro lado, existen componentes como I_x e I_y que son utilizadas para el cálculo de

TP y TD pero también para el cálculo de otros términos como es el caso de TP_x y TP_y. Otro ejemplo, son las componentes $(\overline{u}, \overline{v})$ que fueron utilizadas para calcular el término de TP pero además las mismas componentes son utilizadas para el cálculo de la velocidad (u y v). Por este hecho surge la necesidad de almacenar valores en algún registro temporal durante el proceso. Así después de un tiempo es posible utilizar el contenido de dicho registro. Esto se puede apreciar con mayor claridad en la figura 5.5. Los símbolos ΔT, dentro de la misma figura, significan una señal registrada que equivale a que el resultado o valor del dato se presentará en la salida en el siguiente ciclo de reloj. Las señales LU y LV, mostradas en el diagrama, son las equivalentes de $\overline{u}$ y $\overline{u}$ respectivamente.

Continuando con la descripción para el cálculo de la velocidad, es necesario obtener los términos TP_x y TP_y. Entonces en el siguiente ciclo de reloj se calculan estos términos que son representados en 26 bits en complemento a 2. Paralelamente el término TD es almacenado en un registro temporal ΔT durante un ciclo de reloj ($TD1 <= TD$) para efectuar posteriormente el cálculo de los términos R_x y R_y.

Finalmente, una vez que se conocen los valores de TP_x y TP_y se inicia inmediatamente con el cálculo de los términos R_x y R_y. El cálculo consiste en una división, para cada componentes, entre el término TP_x /TD y TP_y /D. Donde el numerador es de 26 bits con signo y el denominador es de 17 bits sin signo. Esta operación presenta una latencia de 7 ciclos de reloj por lo que se emplea una estructura *pipeline* para poder llevar a cabo el procesado de nuevos píxeles paralelamente.

Una vez calculadas las componentes de la velocidad sus valores deben ser limitados. Valores de velocidad muy grandes indican una discontinuidad o ruido que debe ser suavizado. De esta manera los resultados son truncados para ayudar a converger más rápidamente a un valor real. Es importante aclarar que todas las operaciones son implementadas satisfaciendo la necesidad de bits necesarios para efectuar las operaciones. Entonces un ciclo de reloj después, del cálculo de los términos R_x y R_y, se escriben las componentes de la velocidad horizontal y vertical (u, v) en una memoria de salida `fifo_vel_out`. Las componentes de la velocidad son representadas en 9 bits en complemento a 2.

El control de los diferentes módulos que participan en el cálculo de la velocidad son manejados por el módulo `Ctl_vels`. El bloque de control realiza la lectura de la memoria `fifo_vel_in` y las escribe en el módulo de la velocidad indicándole también que inicie con el procesado de la información. Una vez que los datos son procesados los lee, de la salida del módulo velocidad, y los escribe en la memoria `fifo_vel_out`,

como se muestra en la figura 5.6.

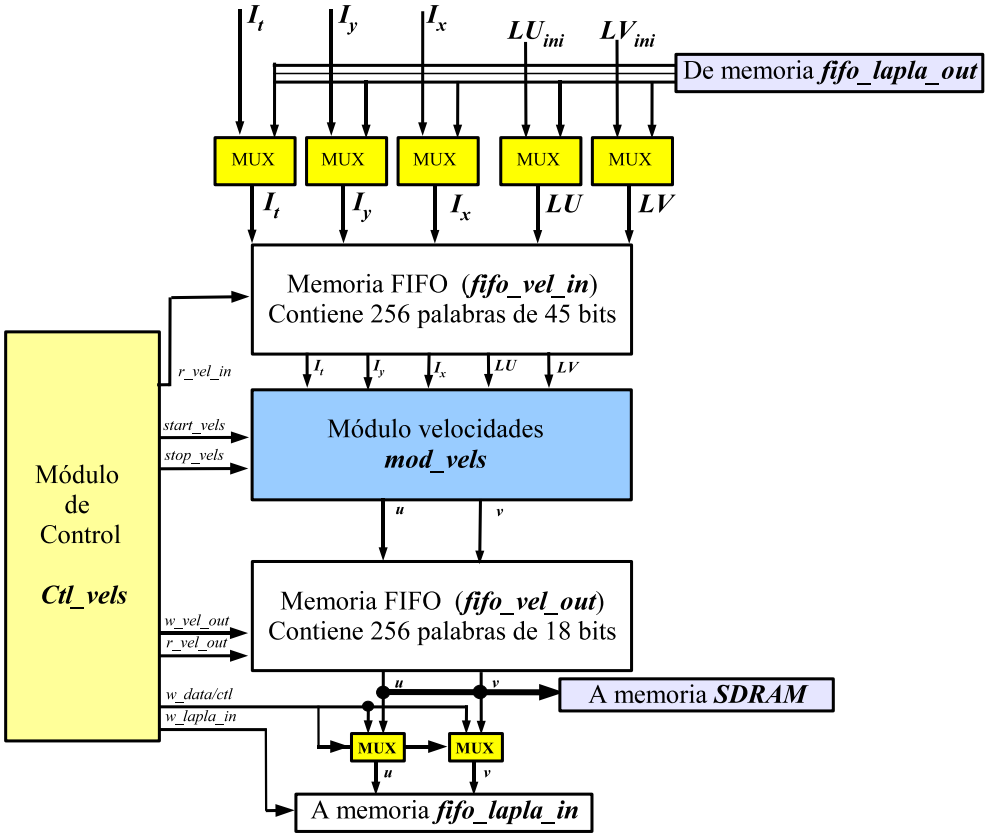


Figura 5.6: *Gestión conjunta para el cálculo de la velocidad y módulos relacionados a esa etapa.*

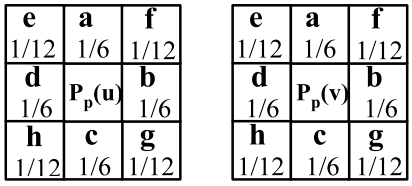
El proceso se ejecuta en paralelo y la transferencia de la información es ejecutada en función de los datos que existan en las memorias FIFO's. Por ejemplo, el módulo `Ctl_vels` detiene el cálculo de la velocidad si la memoria `fifo_vel_in` está vacía o si la `fifo_vel_out` ya está llena. Además de eso `Ctl_vels` realiza otras funciones como es preparar la información para efectuar el siguiente proceso, que consiste en obtener el cálculo de las Laplacianas. Este módulo de control se abordará con más detalle en la sección 5.6.

5.5. Módulo de la Laplaciana

El objetivo de este módulo es realizar un suavizado del campo del flujo óptico obtenido. Este proceso se realiza mediante la convolución de una máscara bidimen-

sional para cada una de las componentes de la velocidad (u, v). La máscara utilizada se puede observar en la figura 5.7 donde muestran distintos pesos según sea la posición del píxel vecino. Para realizar la convolución son necesarios al menos 9 píxeles (3 píxeles consecutivos de 3 renglones consecutivos, de cada imagen). Los 9 píxeles son almacenados en memoria dentro de la misma FPGA, donde se implementa todo el sistema. Con la finalidad de aumentar la frecuencia de procesado, la estrategia de diseño será la misma que la empleada en el módulo del gradiente. Pero en este caso se utilizará una memoria FIFO para almacenar 3 renglones completos, y no dos como fue el caso del módulo del gradiente. Los píxeles de los últimos dos renglones, almacenados en la memoria FIFO, son reutilizados para el cálculo de la laplaciana de los píxeles localizados un renglón abajo. Esto quiere decir que para realizar un futuro cálculo de la laplaciana de un píxel $(i+1, j)$, localizado un renglón abajo del píxel que se está procesando en ese momento (i, j), ya no sería necesario acceder a la memoria principal por ese píxel y los vecinos (i, j), $(i, j+1)$ y $(i, j-1)$. Pues todo el renglón donde se encuentra el píxel a procesar y el renglón previo ya se encuentra en una memoria FIFO localizada dentro de la FPGA. Así pues, sólo será necesario leer un nuevo renglón.

De la misma forma que en el módulo del gradiente, en este módulo se procesan 8 píxeles en paralelo, de 9 bits en complemento a 2, resultando 3 palabras de 72 bits. Las operaciones necesarias para ejecutar este módulo son: 8 sumas y dos divisiones, por píxel y para cada componente $(P_p(u), P_p(v))$. Las operaciones se pueden observar en la figura 5.7.



$$P_p(u) = \overline{u}(u) = \frac{1}{6}(a+b+c+d) + \frac{1}{12}(e+f+g+h)$$

$$P_p(v) = \overline{v}(v) = \frac{1}{6}(a+b+c+d) + \frac{1}{12}(e+f+g+h)$$

Figura 5.7: *Máscara utilizada para llevar a cabo la Laplaciana y las ecuaciones que la representan.*

Al efectuar la convolución de la mascara, de la figura 5.7, es posible predecir dos inconvenientes típicos en este tipo de operación. El primer inconveniente aparece cuando se hace la operación de la máscara en las esquinas o frontera de la imagen. Al intentar aplicar la máscara a un píxel ubicado en una esquina de la imagen, este píxel necesitaría conocer los valores de píxeles vecinos que no existen, como son los píxeles **h**, **d**, **e**, **a**, **f** para el cálculo de $P_p(u)$ o $P_p(v)$. Si analizamos la posición de

esos píxeles (**h**, **d**, **e**, **a**, **f**) estarían localizados en las posiciones $P_p(2,-1)$, $P_p(1,-1)$, $P_p(-1,-1)$, $P_p(-1,1)$ y $P_p(-1,2)$, respectivamente. Haciendo notar que posiciones negativas significan la inexistencia de dicho píxel. El segundo inconveniente es debido a que no se procesa todo el renglón completo en el mismo ciclo de reloj. Entonces se da un hecho similar al primer inconveniente. Al procesar sólo un segmento de la imagen se produce un hecho similar al caso anterior y consiste en la aparición de nuevas fronteras y esquinas por cada segmento de la imagen a procesar. La diferencia al caso anterior es que aquí sí existen los píxeles no conocidos en esas nuevas fronteras de ese segmento de imagen.

La solución del primer inconveniente no existe, pues es un hecho que nunca se podrán conocer los valores vecinos a las esquinas o fronteras de la imagen. Así pues las esquinas o fronteras de la imagen no recibirán un trato especial en su procesado. Esto significa que no se realizará artificio alguno para el llenado de la máscara cuando no se conozcan los valores de los píxeles. Por lo que sólo se procesará la máscara cuando se pueda, sin necesidad de aplicar artificios o mecanismos adicionales para su implementación.

En el caso del segundo inconveniente, que ocurre cuando se procesa un segmento de la imagen de (8×3) píxeles (8 píxeles en paralelo), es posible saber que en el futuro se requerirán los píxeles vecinos para el procesado del siguiente segmento de la imagen. Por esta razón se utilizará una memoria temporal para almacenar las dos últimas columnas de los tres renglones utilizados en el proceso. Ese principio se repetirá durante todo el renglón. Al iniciar el procesado de un nuevo renglón se repetirá nuevamente ese hecho. Para apreciar claramente este caso se presenta en la figura 5.8 un esquema de como se realiza el procesado de los primeros 16 píxeles de la imagen. El procesado de esos 16 píxeles se realiza en dos ciclos de reloj más dos veces la latencia del módulo de la Laplaciana.

En la parte superior y lateral izquierda de la figura 5.8 se indica el número de columna y renglón. Se ve que la primera columna y el primer renglón no serán procesados, la razón es que no existen píxeles vecinos para poder ejecutar correctamente la máscara en la frontera de la imagen, cayendo dentro del primer inconveniente. Por esa razón es necesario iniciar el procesado de los píxeles hasta la columna 2 y el renglón 2. Es posible ver también que siempre que se inicia el procesado de un nuevo renglón sólo será posible procesar 7 de los 8 píxeles que se habían contemplado calcular. Esto debido a que el píxel de la octava columna requiere conocer los datos de los píxeles vecinos ubicados en la columna 9, datos con los que no se cuentan en ese momento. Por tanto significa que para procesar el píxel de la octava columna

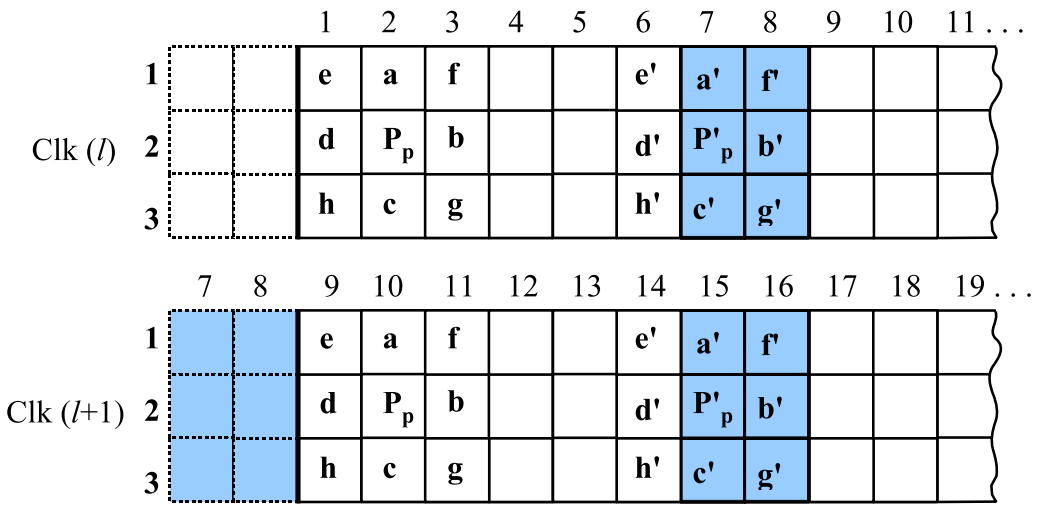


Figura 5.8: *Máscara para el procesado de la Laplaciana de 8 píxeles en paralelo. Se muestran dos segmentos de la imagen indicando la posición de la máscara en los extremos de cada segmento. Se resaltan las columnas utilizadas en el siguiente ciclo de reloj.*

se deberá esperar hasta conocer los valores de los píxeles vecinos (el siguiente segmento de la imagen). Además se entiende que para procesar el píxel de la octava columna es necesario mantener dicha columna y además los valores de los píxeles de la séptima columna. De este modo en el siguiente ciclo de reloj $(l + 1)$ se procesará primero el píxel de la columna 8, que no había sido procesado anteriormente y esa misma columna, la columna número 8 servirá para procesar el primer píxel del nuevo segmento de la imagen (el píxel de la columna 9).

Las operaciones para implementar la máscara son un conjunto de sumas y divisiones, entre los mismos píxeles de la máscara, donde cada píxel tiene un peso determinado en función de la posición, como se mostró en la figura 5.7. En la misma figura se muestra que los píxeles que tienen una distancia de uno (P_{d1}) al píxel que se está procesando son divididos por 6, y aquellos píxeles que tienen una distancia de dos (P_{d2}) son divididos por 12. Las divisiones, en particular la división por 12, representa un pequeño coste adicional hardware y un aumento en el tiempo de procesado, reflejándose en una mayor latencia del módulo. Con el objetivo de reducir el coste hardware y la latencia es posible utilizar un ajuste para realizar sólo un tipo de división, una división por 3. Es claro que es necesario realizar otras dos divisiones una por 2 y otra por 4 para obtener las divisiones que inicialmente fueron contempladas. Esto puede verse reflejado en la ecuación 5.3. Sin embargo, las

últimas dos divisiones debido a que son potencia de dos es posible efectuarlas mediante simples desplazamientos a la derecha. La división por 2, para el término P_{d1}, es realizada con un desplazamiento a la derecha. La división por 4, para el término P_{d2}, es implementada con dos desplazamiento a la derecha. Con esto se logra igualar el tiempo de procesado de ambos pares de divisiones y además el coste hardware se verá reducido al igual que la latencia.

$$(\overline{u},\overline{v}) = \begin{cases} \overline{u}(u) = \frac{a+b+c+d}{6} + \frac{e+f+g+h}{12} = \frac{P_{d1}}{\mathbf{3}\cdot 2} + \frac{P_{d2}}{\mathbf{3}\cdot 4} \\ \overline{v}(v) = \frac{a+b+c+d}{6} + \frac{e+f+g+h}{12} = \frac{P_{d1}}{\mathbf{3}\cdot 2} + \frac{P_{d2}}{\mathbf{3}\cdot 4} \end{cases} \tag{5.3}$$

De esta manera se puede concluir que para el procesado de cada píxel se requieren 8 sumas, dos divisiones por 3 y dos desplazamientos, un desplazamiento de uno y otro desplazamiento de dos. Ambos desplazamientos serán a la derecha y manteniendo el signo y los resultados son representados en 9 bits con signo. Sin embargo, durante las operaciones intermedias fue necesario utilizar registros de 11 bits en los cuales se almacenaron las sumas parciales y los resultados de las divisiones. Pero el resultado final, al concluir el cálculo de las laplacianas, fue posible representarlo en 9 bits en complemento a 2.

En este momento el módulo `Laplaciana` ya realiza la función deseada. Pero es necesario incorporar señales para el control del flujo de los datos, como es el caso de las señales *lapla_ack*, *lapla_stop* y *lapla_start*. El módulo que maneja dichas señales es el `Ctl_lapla`. Entonces la primera función que efectúa el módulo de control, para realizar el cálculo de la laplaciana, es leer la memoria `fifo_lapla_in` y escribir los datos en el módulo Laplaciana y activar la señal *lapla_start* para iniciar el procesado de la información. A su vez e módulo Laplaciana, mediante la señal *lapla_ack* le confirma que existen datos validos en su salida. Así el módulo de control lee los datos de salida y los escribe en la memoria temporal `fifo_lapla_all`. Todo esto se puede observar en la figura 5.9. La memoria temporal `fifo_lapla_all` tiene un propósito muy específico, que es auxiliar para efectuar la discriminación de las componentes que pertenecen a los píxeles que si cambiaron en conjunto con la LUT3. Al ser seleccionadas las componentes, por el módulo `Ctl_lapla`, éstas son escritas en la memoria `fifo_lapla_out`. El módulo `Ctl_lapla` y este proceso se aborda con más detalle en la sección 5.6.

Entonces una vez calculadas las componentes de la Laplaciana $(\overline{u},\overline{v})$, son almacenadas junto con el resto de las componentes necesarias para el cálculo de la velocidad en las futuras iteraciones. De esta manera las componentes LU, LV, I_x, I_y e I_t se al-

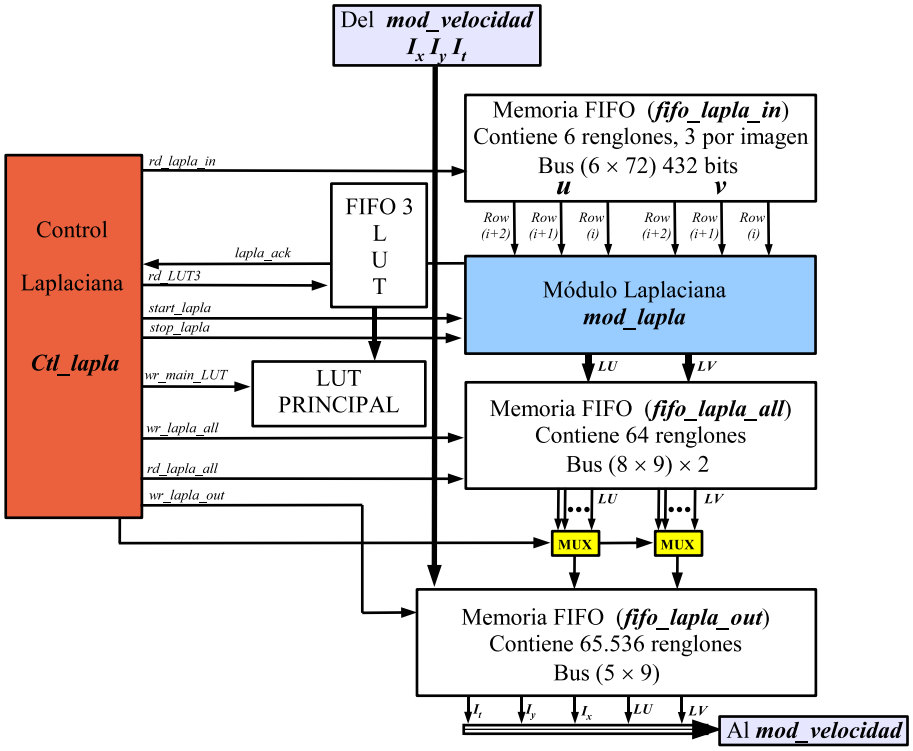


Figura 5.9: *Diagrama a bloque que muestra los módulos que participan en el cálculo de las componentes de la Laplaciana.*

macenan en la memoria `fifo_lapla_out`. La memoria `fifo_lapla_out` está dividida en dos memorias. Una de ellas que almacena LU y LV (`fifo_lapla_outa`) con una capacidad de 1.179.648 bits y la otra que almacena I_x, I_y e I_t (`fifo_lapla_outb`) tiene una capacidad para 1.769.472 bits.

Para la implementación de este módulo fue utilizado el lenguaje de descripción de hardware VHDL, utilizando Quartus II software de Altera y el código VHDL que describe los módulos desarrollados se encuentra en el apéndice A.

5.6. Módulo de control

Hasta este momento se ha descrito el funcionamiento de las unidades que realizan el procesado de las diferentes etapas necesarias para el cálculo del flujo óptico guiado por cambios. Además se ha explicado, de manera general, como fluyen los datos entre los distintos módulos. En esta sección se ahondará más sobre el control y manejo de la información, detallándose los bloques necesarios y las estrategias utilizadas para

mantener siempre un flujo de datos, entre los distintos módulos, lo más constante posible.

Este módulo es sin duda alguna el más complejo de todos los módulos del sistema, pues interviene en todas las etapas y debe ejecutar distintas funciones para realizar el procesado de la información de una forma eficaz. Entre las diferentes funciones se pueden enumerar algunas, como son:

1. La sincronización de todas la etapas que constituyen el sistema.
2. Controlar el flujo de datos y el número de iteraciones durante el cálculo del flujo óptico.
3. Manejar la tabla LUT que será utilizada en diferentes etapas del proceso. Además de conservar la información de los píxeles que mostraron cambios significativos.
4. Gestionar el acceso de lectura/escritura con la memoria principal, localizada en la misma tarjeta de desarrollo donde se encuentra la FPGA utilizada en el diseño.
5. Indicar cuándo un módulo o etapa del sistema debe iniciar o detener su funcionamiento.

Todas las características y acciones de control antes descritas, hacen que el módulo de control tenga una alta complejidad. Adicionalmente, la característica del sistema el cual consiste en una arquitectura de flujo de datos, hace que cada una de las etapas deba ser controlada de manera local. Esto significa que el módulo de control deba ser dividido en varios submódulos, un submódulo para cada una de las etapas en que fue dividido el sistema.

Por otro lado, debido a que es inminente la posibilidad de que exista una función común entre los distintos submódulos, que soliciten ejecutar simultáneamente una misma función, se hace necesario que los submódulos se interconecten a un módulo de control maestro, llamado *árbitro*. El control maestro o *árbitro* se encargará de gestionar cada solicitud de los diferentes módulos y asignar la prioridad correspondiente a cada una de las señales, provenientes de los diferentes submódulos. En particular la señal más común o que presente esta situación será la de escribir o leer en la memoria principal local.

Todo lo expuesto hasta el momento hace pensar que para implementar el módulo de control es necesario utilizar una estrategia de control distribuido. Esto significa que los bloques o módulos deberán tener la capacidad de procesar datos de forma

independiente al resto de las etapas del sistema. Así cada etapa del sistema trabajará en función del flujo de datos que ingrese en cada una de las etapas, en que se dividió el sistema de procesado. La única o únicas restricciones, para los submódulos de control, están en la limitación de la memoria de entrada o salida de cada una de las etapas del sistema. En la figura 5.10 se presenta el diagrama a bloques del sistema aquí propuesto en el que se detallan las señales de control que participan en la ejecución del procesado de imágenes guiado por cambios. En el mismo diagrama se pueden visualizar cada uno de los submódulos de control, las etapas del sistema y las memorias FIFO's de entrada y de salida de cada etapa. De esta manera es posible observar la existencia de 5 submódulos de control: el circuito de control gradiente, el circuito de control gradiente-Laplaciana, el circuito de control velocidades-iteración, el circuito de control Laplaciana y el arbitro/DDR-interfaz.

5.6.1. Circuito Arbitro/DDR-interfaz

Este circuito controla el acceso para escribir/leer en la memoria SDRAM. En realidad es la interfaz-controlador de la SDRAM provista por el *KIT* de desarrollo que contiene el código fuente para realizar transacciones de escritura/lectura sólo entre el bus PCI y la memoria SDRAM. Al incorporar el sistema diseñado fue necesario realizar ciertas modificaciones para poder realizar transacciones de lectura/escritura entre el diseño implementado en la FPGA y la memoria DDR SDRAM. Así mismo mantener la posibilidad de realizar transacciones también entre el bus PCI y la SDRAM. La memoria es del tipo DDR SDRAM, funciona a 166 MHz y se encuentra en la misma tarjeta de desarrollo, donde se localiza la FPGA.

El módulo `Arbitro/DDR-interfaz` realiza la comunicación con la memoria DDR. La comunicación consiste en seguir una serie de protocolos establecidos por las hojas de especificaciones de la memoria, que en este caso es una memoria DDR SDRAM (SODIMM) PC333, con capacidad de 256 Mbytes. El controlador de la memoria DDR es proporcionado por el fabricante de la tarjeta de desarrollo. En este caso es la firma Altera. La información detallada de la interfaz y en particular del controlador se encuentra en [Cor03a] y [Cor05a].

5.6.2. Circuito de control gradiente

Este circuito, llamado `ctl_grad`, se encarga de realizar la petición de lectura a la memoria principal local cuando se plantee iniciar el cálculo del módulo `LUT/Grad`. El

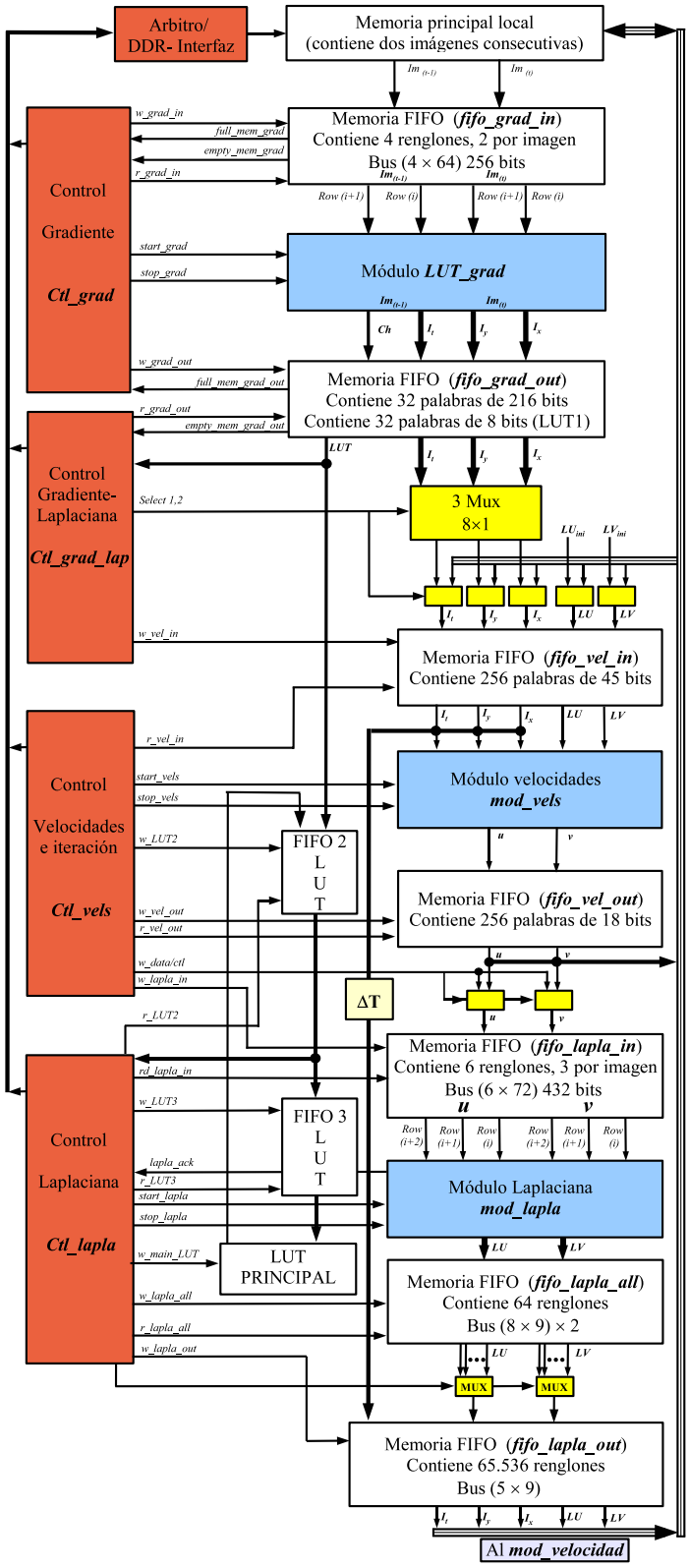


Figura 5.10: *Arquitectura a bloques para el cálculo del flujo óptico guiado por cambios. Se muestran los bloques de control, con las respectivas señales de control, para cada una de las etapas del sistema.*

proceso lo realiza a través del modulo de control `Arbitro_DDR_Interfaz`. Al iniciar por primera vez su funcionamiento deberá controlar que se lean dos renglones consecutivos de cada una de las dos imágenes almacenadas previamente en la memoria SDRAM. Después de eso se deberá leer un solo renglón de cada imagen, hasta concluir con el cálculo del flujo óptico. Para posteriormente iniciar con el procesado de dos nuevas imágenes. Además de realizar la petición de lectura a la SDRAM también escribe los datos en la memoria `fifo_grad_in`, activando la señal *w_grad_in*. La lectura de la SDRAM y escritura en la `fifo_grad_in` se efectúa a una frecuencia de 166 MHz.

La memoria `fifo_grad_in` puede almacenar 4 renglones de 256 píxeles, 2 renglones de cada imagen. Una vez que la memoria `fifo_grad_in` contenga los 4 renglones, ésta informa al módulo de control `ctl_grad` mediante la señal *full_mem_grad*. En ese instante el módulo de control activa la señal *start_grad* indicándole al módulo `LUT_grad` que inicie el proceso del cálculo de los gradientes I_x, I_y, I_t, el cual también realiza la detección de cambios (***ch***) de aquellos píxeles que han mostrado diferencias significativas entre las dos imágenes consecutivas.

El módulo `LUT_grad` procesa 8 píxeles en paralelo para cada uno de los gradientes, que son almacenados en la memoria `fifo_grad_out`. En la misma memoria será almacenada la tabla de cambios LUT. De esta manera la memoria `fifo_grad_out` podrá almacenar 32 palabras de 216 bits cada una, para poder almacenar los 8 píxeles de 9 bits. Cada conjunto de 8 píxeles, que fueron calculados en el módulo `LUT_grad`, es por cada unos de los gradientes I_x, I_y, I_t. También se almacenan, en la misma memoria, 32 palabras de 8 bits cada una, para registrar la información de que píxeles presentaron cambios significativos y que píxeles no presentaron cambio alguno. El módulo `LUT_grad` trabaja a una frecuencia de 33 MHz, por lo tanto la lectura a la memoria `fifo_grad_in` y la escritura en la `fifo_grad_out` se realizan a esa misma frecuencia. La memoria `fifo_grad_out` trabaja a dos frecuencias distintas, una para escritura y otra para lectura.

En caso de que la memoria `fifo_grad_out` se llene, ésta activará la señal *full_mem_grad_out* para que el módulo de control inmediatamente sepa de la situación. El módulo de control a su vez intervendrá y detendrá el funcionamiento del módulo `LUT_grad` mediante la señal *stop_grad*. Este hecho es muy poco probable debido a que el siguiente proceso realiza la lectura de la memoria `fifo_grad_out` a una frecuencia de 166 MHz y además inicia en cuanto exista la primera palabra en la memoria `fifo_grad_out`.

5.6.3. Circuito de control gradiente-Laplaciana

Este circuito, llamado `ctl_grad_lap`, trabaja de forma independiente al igual que el resto de los módulos de control. El módulo de control inicia su actividad en el momento de recibir la señal *empty_grad_out* proveniente de la memoria `fifo_grad_out`. La señal *empty_grad_out* se activará sólo cuando la memoria `fifo_grad_out` tenga datos disponibles.

Esta etapa del proceso, y en particular la función de este módulo de control, resulta ser de las más significativas y relevantes del sistema para el procesado guiado por cambios. La razón es que en esta etapa y este módulo de control es el que se encarga de discriminar los píxeles que presentaron cambios significativos, respecto de aquellos píxeles que se mantuvieron constantes en el tiempo (entre el par de imágenes consecutivas).

Una vez que se sabe que la memoria `fifo_grad_out` posee datos para ser procesados, el módulo de control `ctl_grad_lap` multiplexará los 8 píxeles provenientes de la memoria `fifo_grad_out` a la nueva memoria `fifo_vel_in`, que almacena una sola componente para cada gradiente I_x, I_y e I_t. Lo interesante de la multiplexación es que sólo serán leídas aquellas componentes de los píxeles que presentaron cambios significativos. Este proceso se realizará en función de los datos leídos de la LUT, que una vez utilizados son escritos en una nueva tabla (LUT2). En el caso de que ningún píxel haya presentado cambios significativos, de los 8 píxeles leídos de la memoria, entonces el sistema de multiplexación no escribirá ningún dato en la memoria `fifo_vel_in`. Así la memoria `fifo_vel_in` sólo registra aquellas componentes de los píxeles que han presentado cambios significativos y que serán los únicos procesados por el sistema, en las futuras etapas. Todo este proceso se ejecuta a 166 MHz y paralelamente al cálculo de los gradientes espacio-temporal.

Paralelamente el circuito de control `ctl_grad_lap` iniciará los valores de LU y LV, siempre y cuando sea la primera iteración. Estos valores son necesarios para realizar el cálculo de las velocidades. En caso de que se esté realizando una iteración distinta a la inicial, los valores de las Laplacianas y de los gradientes son leídos de la memoria `fifo_lapla_out`. Los datos leídos que serán escritos en la memoria `fifo_vel_in` son de un tamaño de palabra de 45 bits. La memoria tiene una capacidad máxima de 256 palabras, dimensión igual a la de un renglón de las imágenes utilizadas en este trabajo. La siguiente etapa del sistema leerá de esta memoria los datos, LU, LV, I_x, I_y e I_t, para llevar a cabo el cálculo de las componentes de la velocidad.

5.6.4. Circuito de control velocidades-iteraciones

Este circuito, llamado `ctl_vels`, inicia su proceso cuando la memoria `fifo_vel_in` contiene datos a procesar, de esta manera el módulo realiza la lectura de la memoria activando la señal *r_vel_in*. Una vez que lee los datos, éstos son escritos en el módulo de velocidades activando la señal *start_vels* para que se inicie el procesado de la información. Una vez que ingresan los datos y después de 9 ciclos de reloj, que es la latencia del módulo `mod_vels`, se activa la señal *vels_ack* indicándole al circuito `ctl_vels` que existe datos validos en la salida del módulo `mod_vels`. De esta manera los datos son leídos y escritos en la memoria `fifo_vel_out`, activando la señal *w_vel_out*.

Posteriormente, si es la última iteración a efectuar el módulo de control `ctl_vels` hacer una petición de escritura al módulo *Arbitro_DDR_Interfaz*. Así lee los datos de la memoria `fifo_vel_out`, activando la señal *w_vel_out*, y los escribe en la SDRAM. Además de esto también se lee la LUT PRINCIPAL y es escrita en la SDRAM.

En caso de que no sea la última iteración, el circuito de control `ctl_vels` realiza otra operación que consiste en realizar la reconstrucción de las imágenes que contienen las componentes de las velocidades, con el fin de efectuar el cálculo de la Laplaciana. El proceso no es simple debido a que consiste en la convolución de una máscara bidimensional y para esto se requieren todos los píxeles y no sólo aquellos que fueron procesados. Se debe entender que los píxeles procesados fueron sólo los que presentaron cambios y que serán por consiguiente una fracción de toda la información. Por este motivo al momento de leer los datos de la memoria *fifo_vel_out* y escribirlos en la nueva memoria *fifo_lapla_in* deberán ser reconstruidas las imágenes de las componentes de la velocidad.

Para la reconstrucción de estas imágenes, de la imagen u y la imagen v, se utiliza la tabla LUT2. En función de los datos leídos en la LUT2 se sabrá si los datos leídos de la memoria *fifo_vel_out* se escribirán en la memoria *fifo_lapla_in* o en su caso se escribirán datos para reconstruir la imagen. Este proceso se lleva a cabo con la señal *w_data/ctl* que maneja un circuito lógico para este proceso. La memoria *fifo_lapla_in* tiene la capacidad de almacenar 6 renglones, 3 renglones por cada imagen. Por otro lado, los datos leídos de la LUT2 serán escritos en una nueva tabla LUT3.

Resumiendo, el objetivo de este módulo de control es realizar el cálculo de las componentes de la velocidad (u, v), en caso de que sea la última iteración escribir las componentes de la velocidad en la memoria SDRAM. En caso de que no sea la última iteración, entonces se llevará a cabo la reconstrucción de las dos imágenes

de la componente de velocidad (horizontal y vertical) para poder realizar la última etapa del procesado de la información. La última etapa, para el cálculo del flujo óptico, es el suavizado de la imagen, que consiste en realizar la convolución de una máscara bidimensional, denominada Laplaciana.

5.6.5. Circuito de control Laplaciana

El módulo de control previo `ctl_vels` dejó disponibles los datos en la memoria `fifo_lapla_in` para realizar el cálculo de la Laplaciana. El submódulo de control Laplaciana, llamado `ctl_lapla`, tiene como objetivo realizar ese proceso mediante la convolución de una máscara bidimensional y escribir el resultado de la operación en la memoria `fifo_lapla_all`.

Para realizar este proceso, el circuito de control inicia su funcionamiento cuando la memoria `fifo_lapla_in` hace saber, con la señal *full_lapla_in*, que contiene 3 renglones de cada imagen para iniciar el procesado de la información. En este momento el circuito `ctl_lapla` indica al módulo Laplaciana `mod_lapla` que puede iniciar el procesado de los datos, con la señal *start_lapla*. A su vez el resultado de efectuar el cálculo de la laplacianas será escrito en la memoria `fifo_lapla_all`. Todo este proceso se realiza a una frecuencia de 33 MHz.

Simultáneamente, pero a una frecuencia de 166 MHz, se está leyendo la memoria `fifo_lapla_all` y la LUT3 para realizar una discriminación de las componentes que pertenecen a los píxeles que presentaron cambios significativos. De esta manera, y debido a que el cálculo de la laplaciana fue de 8 píxeles en paralelo, es necesario multiplexar la información para almacenar en la memoria `fifo_lapla_out` una sola componente de LU, LV, I_x, I_y e I_t. Se hace de esta forma debido a que al leer una palabra de la memoria `fifo_lapla_out` es suficiente para proporcionar todos los datos necesarios para el cálculo de la velocidad en las futuras iteraciones. La dimensión de la memoria `fifo_lapla_out` es de 65.536 palabras de 45 bits, necesaria para las 5 componentes de 9 bits cada una. Por otro lado las palabras leídas de la LUT3 son almacenadas en una LUT PRINCIPAL, que posee un tamaño de 32.768 bits, lo que equivale a 4.096 palabras de 8 bits. La LUT PRINCIPAL es leída y escrita en la SDRAM por el módulo `ctl_vels`.

Capítulo 6

Simulación y Síntesis

6.1. Introducción

Durante el proceso de diseño de un sistema hardware se presenta un problema fundamental que no existe en los procesos de diseño de sistemas software. Este problema es el alto coste de ciclo diseño-prototipación-testeo-vuelta a empezar, ya que el coste del prototipo suele ser comúnmente bastante elevado. Por tal razón es necesario reducir este ciclo de diseño para sólo incluir la fase de prototipación en la parte final de proceso. Para ello se introduce la fase de simulación y comprobación de circuitos utilizando herramientas de diseño asistido por computadora (CAD). De esta manera no es necesario realizar físicamente un prototipo para comprobar el funcionamiento del circuito, si no hasta la parte final de diseño, economizando así el ciclo de diseño.

En el ciclo de diseño hardware las herramientas de CAD están presentes en todos los pasos. La simulación permite visualizar los resultados y evaluar el rendimiento del sistema o de sólo una fracción del sistema, si así se requiere. De esta forma, mediante el uso de simulaciones locales o globales es posible realizar ajustes buscando mejorar el rendimiento y obtener un sistema final óptimo. De esta manera es importante hacer notar que la simulación es una parte, dentro del proceso de diseño, sumamente importante y determinante para ahorrar tiempo de diseño. Una vez que se ha verificado el funcionamiento correcto del circuito (simulación funcional) se procede a realizar una segunda simulación llamada *timing simulation*. En la simulación digital se tiene en cuenta los retrasos en la propagación de las señales digitales.

Una vez superadas las etapas de simulación, el diseño avanza a la fase de im-

plantación o fabricación. En este caso debido a que se está utilizando un lenguaje de descripción de hardware, en particular el VHDL, utilizando una FPGA para la implantación del diseño, es necesario realizar la **síntesis** del circuito diseñado.

La síntesis de un circuito, a partir de una descripción VHDL, consiste en reducir el nivel de abstracción de la descripción del circuito hasta convertirlo en una definición puramente estructural, cuyos componentes son los elementos básicos utilizados por una determinada biblioteca o bien por la herramienta utilizada para efectuar la compilación y el sintetizado. La herramienta utilizada deberá incluir, entre los componentes que soporta el dispositivo específico en el cual se va a programar el diseño. En este caso la herramienta utilizada para compilación y síntesis es Quartus II software v.5.1.0 de Altera y el dispositivo es de la familia Stratix, la EP1S60F1020C6 de Altera.

Por otro lado, para casos donde la función del diseño hardware es procesar imágenes se torna forzoso manejar una gran cantidad de datos. El hecho de manejar imágenes conlleva la realización de un procesamiento masivo de información y por consiguiente las simulaciones no son una tarea nada fácil pues se deberá manejar una gran cantidad de datos o vectores de prueba. Esto hace necesario utilizar herramientas más sofisticadas para realizar la simulación. Actualmente las herramientas para realizar la simulación y síntesis de circuitos digitales están equipadas con utilidades para crear señales de estímulos pero ellas están limitadas para aplicaciones que requieren una gran cantidad de datos. Tradicionalmente son utilizados vectores de prueba, durante las simulaciones en diseños de sistemas de no muy grande envergadura, sin embargo, para simulaciones de sistemas de procesamiento de imágenes, y sobre todo en las últimas etapas del diseño (simulaciones globales), existe la necesidad de manejar una densidad de datos que son difícilmente manejables con vectores de prueba sencillos.

Para simulaciones muy complejas como es el procesamiento de imágenes, la densidad de datos utilizados y procesados, dificulta el manejo de la información y la identificación de errores existentes. Por esta razón se propuso llevar a cabo la simulación del sistema en varias etapas. En una primera etapa se realiza una simulación local, en la segunda etapa se realiza una simulación completa del sistema diseñado y propuesto en este trabajo. Finalmente en la última etapa de simulación se involucra un banco de pruebas o *test-bench* por lo que se puede considerar que es el nivel de simulación superior en el cual se consideran simulación de elementos externos como las posibles señales generadas por un computador personal. En cada una de las tres etapas de simulación se sabe a priori qué cantidad de datos se van a procesar y de

esta manera se elige tanto la herramienta adecuada para la simulación y el tipo o tipos de vectores de prueba a utilizar.

Así pues, el proceso de simulación se realiza en tres etapas de simulación crecientes, donde cada etapa tiene un objetivo específico:

1. **Simulación local:** Consiste en realizar la simulación de cada uno de los módulos o bloques de manera independiente. El objetivo es visualizar y mejorar los tiempos de ejecución, conocer el área requerida por cada módulo en el dispositivo utilizado, y optimizar los recursos hardware necesarios. Todo esto procurando elegir adecuadamente el tipo de arquitectura utilizada, ya sea segmentada, combinacional, etc., para dar una solución óptima al diseño. Para este tipo de simulación se utilizaron pequeños vectores de prueba. Para la simulación sólo se utilizó Quartus II software. MATLAB fue utilizado para comparar los resultados obtenidos en la simulación, el cual maneja aritmética entera, contra los resultados obtenidos por un sistema que procese la información en coma flotante. Por último se realiza una síntesis de cada módulo diseñado para tener un conocimiento aproximado de la cantidad necesaria de recursos hardware para la implementación del sistema dentro de la FPGA.
2. **Simulación global:** En esta simulación se pretende evaluar todos los módulos diseñados de manera conjunta, además de las memorias utilizadas en el sistema. El objetivo es monitorear la sincronización entre los módulos y el acceso a la memoria externa SDRAM, en la cual se almacenan las dos imágenes consecutivas a ser procesadas. Por otro lado se realiza la compilación y síntesis para conocer el espacio utilizado en la FPGA por todo el sistema. Así pues se simularán los bloques previamente evaluados, las memorias FIFO's utilizadas y el controlador de la SDRAM como un solo sistema. Para esta simulación se utilizan vectores de prueba más elaborados, incluso imágenes reales de prueba comúnmente utilizadas. Por lo tanto las herramientas de simulación utilizadas son más robustas, como es el caso de ModelSim-Altera, además de Quartus II software para la síntesis y MATLAB para desplegar y realizar cálculos en coma flotante.
3. **Simulación final con un banco de pruebas:** En esta simulación se incluye un banco de pruebas o *Test-bench* para poder llevar a cabo la simulación final del sistema de visión. Sin embargo, en esta etapa es necesario integrar un nuevo bloque al diseño. Este nuevo bloque es una función que sirve para la comunicación con el bus PCI llamado **MegaCore PCI**. En si es la interfaz entre el banco de pruebas (que simula al computador y todas sus señales de control)

con el sistema diseñado. La incorporación del **MegaCore PCI** y el banco de pruebas permite llevar a cabo una simulación en la que se incluya el resto de los componentes que conforman a un sistema de visión. Entre los componentes están: el computador principal, el bus PCI de interconexión, el controlador y la memoria SDRAM, y el sistema diseñado para procesar las imágenes (obtención del flujo óptico). Las herramientas utilizadas para la comprobación, simulación y síntesis fueron: ModelSim-Altera, Quartus II software y MATLAB. La síntesis fue realizada para la FPGA, de la familia Stratix, EP1S60F1020C6 de Altera.

La utilización de varias herramientas durante el proceso de simulación tiene la finalidad de analizar y visualizar de manera más cómoda y rápida los resultados de las imágenes de salida del sistema diseñado. Así pues se utilizaron básicamente tres herramientas para realizar la simulación, síntesis y comprobación de los datos, las cuales son: Quartus II software, ModelSim-Altera y MATLAB.

Un diagrama esquemático que describe los bloques a simular en cada una de las tres etapas es mostrado en la figura 6.1. En la figura se puede ver claramente que para procesar las imágenes, el sistema está constituido por 4 bloques, `LUT/Grad, Velocidad, Laplaciana y Sistema de control`. Cada uno de los bloques es simulado de forma independiente. Una vez realizada esa etapa de simulación se procede a realizar una simulación global del sistema diseñado, lo cual incluye el sistema de procesamiento, memorias FIFO's utilizadas para almacenar datos temporales entre los módulos y finalmente un bloque para controlar la SDRAM. Una vez superada la simulación global, se realizó una última simulación que incluye el uso de un banco de pruebas y adicionalmente un bloque para llevar a cabo la interfaz entre el banco de pruebas y el sistema para el procesamiento de las imágenes. Este bloque adicional es incluido en el diseño del sistema de visión y consiste en el **MegaCore PCI**.

6.2. Simulación de los módulos

La primera etapa de simulación consiste básicamente en evaluar el rendimiento de cada uno de los módulos diseñados de forma independiente y conocer la cantidad de recursos utilizados por la FPGA empleada. Todo esto se lleva a cabo mediante simulaciones funcionales, digitales y el sintetizado de los bloques diseñados. Así pues, se puede decir que la finalidad de esta etapa de simulación es realizar los ajustes necesarios para obtener la máxima frecuencia de procesado, optimizar lo mejor posible el código de descripción de hardware (VHDL) para reducir el coste

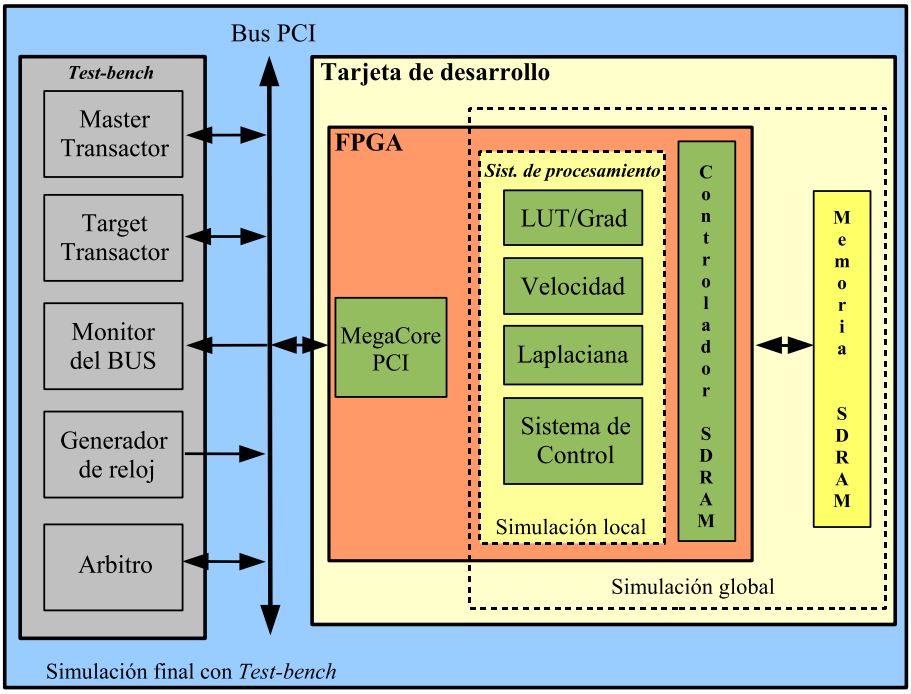


Figura 6.1: *Diagrama a bloques de las tres etapas de simulación y de cada uno de los módulos que lo conforman.*

hardware necesario para su implementación. Además, en esta etapa de simulación, también se busca detectar alternativas de implementación de los algoritmos con objetivos claros como la paralelización de las funciones a implementar, con el mínimo de puertas lógicas necesarias para el diseño, y respetando la frecuencia a la que se plantea trabajar que es a 33 MHz. Otro objetivo específico es el utilizar de manera eficiente la FPGA empleada, la cual cuenta con bloques especiales como son: DSP's, PLL's, varios tipos de memoria RAM, DLL's, multiplicadores embebidos, entre otros componentes. En el cuadro 6.1 se muestran las características de la FPGA utilizada, la Stratix EP1S60F1020C6 de Altera.

Dentro de las características de los elementos que conforman la FPGA, existe la posibilidad de configurar de distintas formas algunos de los elementos. Por ejemplo, las memorias pueden ser configuradas de distintos modos, como es: modo puerto simple o puerto dual, utiliza 1 ó 2 frecuencias de reloj, las salidas pueden ser registradas o no registradas, bits de paridad y registros con desplazamiento para las aplicaciones con los DSP. Y lo más notable es que los registros de desplazamiento son totalmente independientes de las celdas lógicas existentes, pues los registros contemplan esta posibilidad o modo de aplicación y por lo tanto poseen recursos pro-

Característica	**No. de elementos o dato**
LEs	57.120
M512 RAM (32 × 18 bits)	574
M4K RAM (128 × 36 bits)	292
M-RAM (4K × 144 bits)	6
TOTAL bits de RAM	5.215.104
DSPs	18
Multiplicadores embebidos	144
PLLs	12
Terminales de I/O	1022
Voltaje de alimentación	1,5 V
Tecnología de fabricación	0,13μm

Cuadro 6.1: *Características de la FPGA EP1S60F1020C6, de la familia Stratix de Altera.*

pios y en ningún momento se utilizan recursos combinacionales adicionales, según el manual de Stratix [Cor04] y [Cor05d].

Los módulos simulados en esta etapa son los descritos dentro del bloque **simulación local** el cual se encuentra en la figura 6.1. El primer paso, dentro de esta etapa de simulación, consiste en realizar una simulación funcional de los módulos. Con la simulación funcional sólo se comprueba el funcionamiento de los circuitos digitales. Es decir a partir del comportamiento lógico de sus elementos (sin tener en cuenta problemas eléctricos como retrasos, etc.) se genera el comportamiento del circuito frente a unos estímulos dados. Posteriormente se realiza una simulación digital (*timing simulation*) la cual tiene en cuenta los retrasos en la propagación de las señales digitales. Las simulaciones mostradas en las siguientes secciones y etapas del proceso de simulación serán solamente simulaciones digitales (*timing simulation*), a menos que se indique que son simulaciones funcionales.

Los estímulos dados para llevar a cabo la simulación son generados de manera manual. Se utilizan una serie de vectores de prueba sencillos y fáciles de generar, que básicamente son un conjunto de valores numéricos seleccionados. La intención de utilizar los vectores de prueba es evaluar el error numérico resultante de las operaciones que se ejecutan en los distintos módulos. De manera paralela los mimos vectores de prueba son introducidos y evaluados en un programa desarrollado en MATLAB. Esto es con la finalidad de comparar ambos resultados, los resultados

obtenidos en la simulación hardware y los obtenidos en MATLAB.

Es importante resaltar que el resultado obtenido en la simulación del diseño hardware es producto de un sistema que utiliza aritmética entera y por otro lado los resultados obtenidos utilizando el software de MATLAB es producto de operaciones que utilizan aritmética de coma flotante. Así pues, es posible asegurar que muchas veces existirán diferencias entre los resultados, para un valor específico de entrada, sin embargo, el propósito de esto es reducir lo menor posible dentro de la arquitectura hardware diseñada la diferencia existente y además poder visualizar la magnitud del error.

El diseño y simulación fue desarrollado en el lenguaje de descripción de hardware VHDL utilizando la herramienta de Altera Quartus II software v.5.1.0. El código VHDL, básicamente comportamental, de todos los algoritmos que se han diseñado se muestran en el apéndice A.

6.2.1. Simulación del módulo `LUT_grad`

Durante la simulación de este módulo los vectores de prueba utilizados fueron un conjunto de datos seleccionados con el objetivo de cubrir posibles valores que puedan presentar alguna inconsistencia en su procesado, como puede ser la división por cero, resultados negativos, etc. Los 8 píxeles deben ser tomados de dos imágenes consecutivas, cuatro píxeles de cada imagen, los cuales están identificados por las letras *a, b, c, d, e, f, g* y *h*. En este caso se asignaron los valores de los píxeles. Para una identificación más fácil, de los píxeles y del nombre que toma cada uno de ellos como variable en la simulación, se presenta la figura 6.2, en la cual se muestra la distribución de los píxeles y como se reflejan los píxeles en las funciones realizadas.

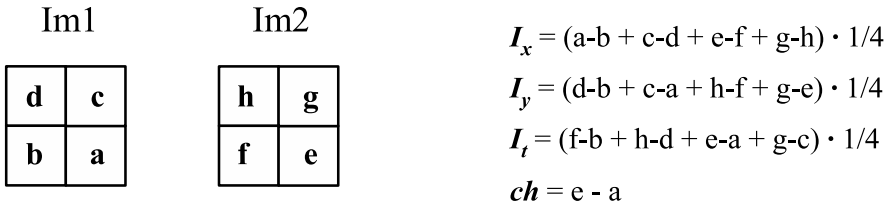


Figura 6.2: *Distribución de los píxeles y las funciones para procesar el módulo `LUT_grad`.*

Cada píxel es de 8 bits sin signo, lo que significa que puede tomar 256 posibles valores. Las salidas I_x, I_y, I_t podrán ser negativas y tomarán valores que van desde

-256 hasta +255, ello significa que se requiere un registro de 9 bits para representar el valor de salida [8..0]. Para obtener cada una de estas salidas es necesario almacenar temporalmente la suma de los 8 píxeles en un registro de 11 bits ([10..0]) y posteriormente realizar la división entre 4. Debido a que la división entre 4 es potencia de dos, entonces es posible implementarla mediante dos desplazamiento a la derecha, conservando el signo, o bien tomando sólo los 9 bits más significativos ([10..2]) de su registro de salida de 11 bits ([10..0]).

En la simulación, mostrada en la figura 6.3, es posible observar la entrada de los 8 píxeles que son procesados y las componentes de salida I_x, I_y e I_t. También se observa la salida *ch*, de tamaño 1 bit, la cual indica si existe una diferencia entre el píxel *e* y el píxel *a*. La magnitud de la diferencia se compara con un umbral establecido previamente, en este caso el umbral es t_h= 1. Si la magnitud es mayor al umbral entonces *ch*=1, indicando que se presento un cambio, en otro caso *ch*=0.

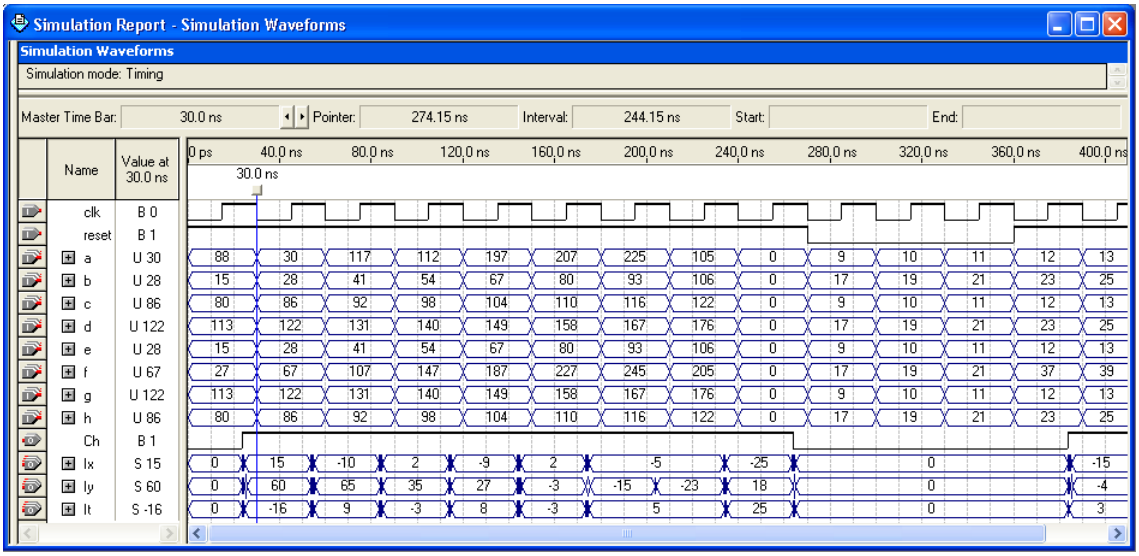


Figura 6.3: *Simulación del módulo* `LUT_grad`*, del cual se obtiene una sola componente de los gradientes I_x, I_y e I_t y una componente* ch.

En el cuadro 6.2 se presentan los datos de salida obtenidos por la simulación del circuito `LUT_grad`, el cual realiza las operaciones utilizando aritmética entera. Los datos difieren muy poco, e incluso algunos no difieren nada, de los valores que fueron calculados por el software de MATLAB, que utiliza aritmética de coma flotante.

De esta manera es posible observar en la figura 6.3, que por cada 8 píxeles procesados se obtienen una componente de I_x, una de I_y, una de I_t y un bit *ch*. Eso significa que si se pretenden procesar más componentes es necesario que ingresen

Entradas (a,b,c,d,e,f,g y h)	**MATLAB**			**Quartus II**		
	I_x	I_y	I_t	I_x	I_y	I_t
88, 15, 80, 113, 15, 27, 113, 80	15	60	-15	15	60	-16
30, 28, 86, 122, 28, 67, 122, 86	-9	65	9	-10	65	9
117, 41, 92, 131, 41, 107, 131, 92	2	35	-2	2	35	-3
112, 54, 98, 140, 54, 147, 140, 98	-8	27	8	-9	27	8
197, 67, 104, 149, 67, 187, 149, 104	2	-3	-2	2	-3	-3
207, 80, 110, 158, 80, 227, 158, 110	-5	-14	5	-5	-15	5
225, 93, 116, 167, 93, 245, 167, 116	-5	-22	5	-5	-23	5
105, 106, 122, 176, 106, 205, 176, 122	-25	18	25	-25	18	25

Cuadro 6.2: *Comparación de los valores obtenidos utilizando aritmética de coma flotante, mediante el software MATLAB, y los valores obtenidos mediante la simulación de la arquitectura hardware, que utiliza aritmética entera. La simulación fue realizada con Quartus II software.*

más píxeles. Este hecho hace pensar que si el objetivo es hacer más eficiente la arquitectura es necesario procesar la mayor cantidad de píxeles como sea posible mediante una arquitectura paralela. La forma más sencilla de realizar este propósito es duplicar la misma arquitectura las veces que sean necesarias. Sin embargo, existe una característica intrínseca en el proceso, explicada en la sección 5.3, que hace posible utilizar unos píxeles para calcular dos componentes de I_x, I_y e I_t y dos bits *ch* simultáneamente. Eso permite ahorrar tiempo de acceso a memoria y además tener una reducción de recursos hardware. Por ejemplo, para procesar dos componentes I_x son necesarios 16 registros de 8 bits y 16 accesos a memoria, uno por cada bit sin considerar tiempos para el direccionamiento o petición. Sin embargo, el mismo proceso es posible realizarlo con 12 registros de 8 bits y 12 accesos a memoria, como se explicó en la sección 5.3, reduciendo así en un 25 % los registros necesarios y en un 25 % el tiempo necesario para acceder a memoria y leer los píxeles utilizados en el procesado de las dos componentes I_x.

Por lo antes expuesto, es posible formular un diseño del módulo `LUT_grad` que pueda procesar 8 componentes simultáneamente. También se considera una reducción de los recursos hardware utilizados. Esta consideración se debe a que durante la integración de 8 bloques, similares al primer módulo propuesto, existe un solapamiento de varios píxeles entre los módulos. La simulación de la arquitectura diseñada, que realiza el procesamiento de 8 componentes simultáneamente, es presentada en

la figura 6.4. En este caso los vectores de prueba son un poco más elaborados, que la simulación anterior, pero siguen el mismo principio de prueba sobre la evaluación de valores que pudiesen dar algún tipo de inconsistencia.

Los vectores de prueba utilizados se representan en hexadecimal debido a que con 2 dígitos en hexadecimal es posible representar 8 dígitos binarios o bien un píxel de la imagen. Por esta razón no se muestran las componentes individuales de entrada, como en el caso de la simulación efectuada en la figura 6.3, en su caso son utilizados 4 vectores de prueba, dos vectores (*q11* y *q12*) representan a dos renglones de la imagen 1 y los otros dos vectores (*q21* y *q22*) representan a dos renglones de la imagen 2. Los datos de salida del bloque son: Ix_0 ... Ix_7, $Iy_0..Iy_7$, $It_0..It_7$ y $ch_0..ch_7$ que representan las 8 salidas de 9 bits de cada uno de los tres gradientes obtenidos y 8 bits del bloque de detección de cambios, sin embargo, dada la cantidad de vectores de salida fue imposible mostrar alguna de las salidas *ch*.

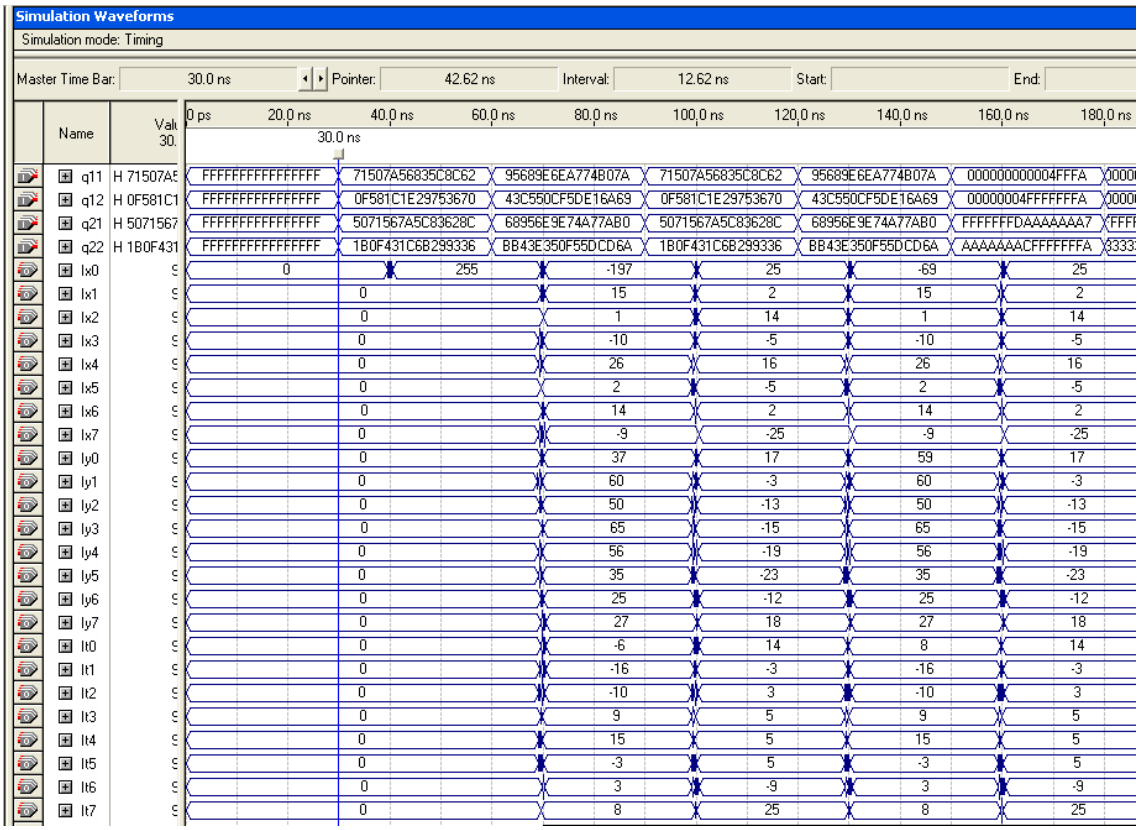


Figura 6.4: *Simulación del módulo* **`LUT_grad`** *que procesa 8 componentes, de* I_x, I_y, I_t *y* ch, *simultáneamente.*

Durante el proceso de simulación digital también se realizó la síntesis del circuito

Parámetros y opciones para la compilación, análisis y síntesis	**Tipo**
Run Assembles during compilation	√
Auto DSP Block Replacement	√
Auto ROM Replacement	√
Auto RAM Replacement	√
Auto RAM Block Balancing	√
Auto Shift Register Balancing	√
DSP Block Balancing:	Auto
State Machine Proccessing:	Auto
Restructure Multiplexers:	Auto
Power Play Optimization:	Normal compilation
Optimization Technique	Balanced (Speed-Area)
HDL Message Level:	Level2

Cuadro 6.3: *Parámetros y opciones para la compilación, análisis y síntesis.*

obteniendo los siguientes datos en el reporte de compilación: utiliza un total de 1.921 elementos lógicos de 57.120, el peor tiempo (*clock setup time*) t_{su}=19,329ns, el peor tiempo (*clock to output delay*) t_{co}=10,544ns y la frecuencia máxima a la que puede trabajar el circuito es de 73,75 MHz. Para todos los casos, de los módulos diseñados en esta sección, los parámetros y opciones para la compilación, análisis y síntesis son mostrados en la tabla 6.3.

6.2.2. Simulación del módulo `velocidad`

El módulo `velocidad` es sin duda la etapa más compleja del sistema aquí desarrollado y la que requiere más recursos hardware. Al igual que el resto de los módulos, los cálculos realizados son hechos utilizando aritmética entera. El utilizar aritmética entera para la implementación de los módulos tiene como ventaja la optimización de recursos hardware y un incremento de la velocidad del sistema. La desventaja es una disminución de la precisión de los datos obtenidos, aunque hay que considerar que siempre la reducción de la precisión es relativa al diseño.

Como se explicó en la sección 5.4 el módulo `velocidad` consiste en una serie de sumas, productos y una división para cada componente de la velocidad. La división es la operación más costosa de este módulo y se ejecuta sobre una arquitectura

segmentada (*pipeline*). La latencia del circuito divisor es de 6 ciclos de reloj y permite obtener las componentes horizontales y verticales de los vectores de flujo óptico. El implementar el circuito divisor mediante una arquitectura segmentada tiene la finalidad de aprovechar el tiempo de latencia. De esta manera una vez iniciada la operación, y después de 6 ciclos de reloj, el circuito divisor es capaz de procesar una división por cada ciclo de reloj. De esta manera la latencia total del circuito `velocidad` es de 8 ciclos de reloj, como puede observarse en la simulación mostrada en la figura 6.5.

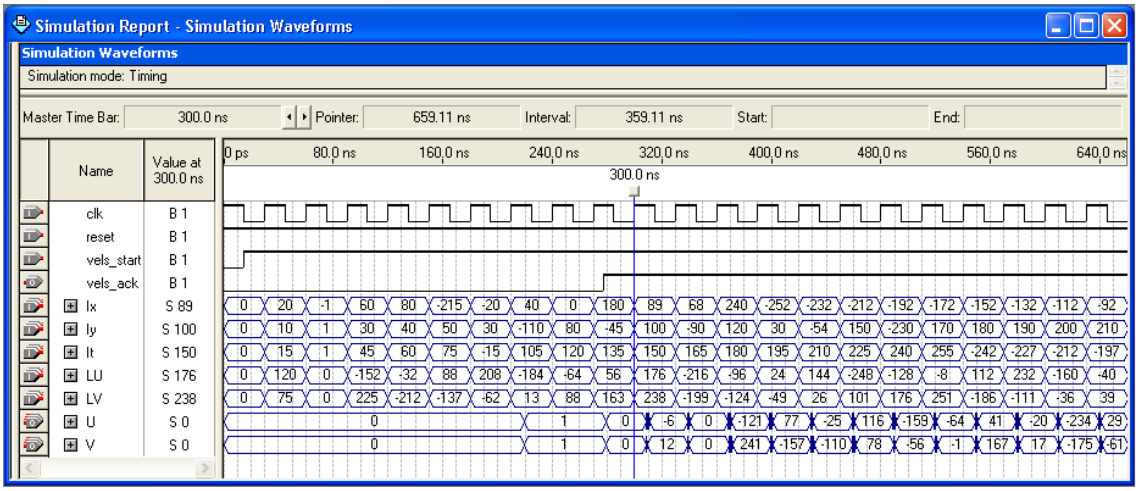


Figura 6.5: *Simulación del módulo* **`velocidad`** *que obtiene las dos componentes de la velocidad simultáneamente.*

En la simulación del módulo se muestran las 5 entradas (I_x, I_y, I_t, LU y LV) y las dos componentes de salida (U, V) del vector de flujo óptico. También se muestran las señales de control y de reloj utilizadas en el módulo. La señal *vels_start* indica el momento en que se debe de iniciar el cálculo el módulo velocidad y es habilitada por un módulo de control que gestiona dicha operación. La señal *vels_ack* indica que el resultado de la operación se encuentra en el registro de salida del módulo. La señal de reset inicializa el módulo y es activa en nivel bajo. También se muestra la señal de reloj *clk* la cual trabaja a una frecuencia de 33 MHz.

Los vectores de prueba utilizados para la simulación y su resultado permite observar la diferencia entre los cálculos efectuados con aritmética de coma flotante (MATLAB) y los cálculos realizados con aritmética entera, obtenidos por la simulación del módulo `velocidad` mediante Quartus II software. En el cuadro 6.4 se muestran los valores obtenidos mediante MATLAB y Quartus II. Es posible observar que existen pequeñas diferencias, y en algunos casos ninguna diferencia, entre los

Entradas					**Salidas**			
					MATLAB		**Quartus**	
I_x	I_y	I_t	LU	LV	U	V	U	V
20	10	15	120	75	-6,60	11,7	-6	12
-1	1	1	0	0	0,50	-0,5	0	0
60	30	45	-152	225	-121	240,5	-121	241
80	40	60	-32	-212	77,80	-157,1	77	-157
-215	50	75	88	-137	-25,38	-110,63	-25	-110
-20	30	-15	208	-62	115,15	77,27	116	78
40	-110	105	-184	13	-158,64	-56,73	-159	-56

Cuadro 6.4: *Valores calculados en coma flotante, utilizando el software MATLAB, y los valores obtenidos por Quartus II software, de la arquitectura hardware diseñada, la cual utiliza aritmética entera.*

valores obtenidos mediante MATLAB y Quartus II. Esto indica que la utilización de aritmética entera para efectuar los cálculos es una opción adecuada y viable.

Adicionalmente a la simulación, el módulo es sintetizado para lo cual se debió indicar la familia de la FPGA utilizada y específicamente el tipo de dispositivo utilizado, en este caso fue un Stratix EP1S60F1020C6. Además, durante el diseño se crean las funciones necesarias especificando que tipo de componentes se van a utilizar. Se debe recordar que el dispositivo aquí utilizado cuenta con una cantidad determinada de DSP's, memoria, LE's, PLL's, etc., que pueden ser utilizadas para aprovechar de manera eficiente el dispositivo. En este caso se utilizaron dos divisores **LPM**, dos multiplicadores **LPM** y dos multiplicadores con sumadores integrados **ALTMULT_ADD**.

Así pues, al realizar la compilación, análisis y síntesis se obtiene que se utilizan 1.613 elementos lógicos de un total de 57.120, 8 DSP's (9×9 bits) de un total de 144 y 145 bits de memoria de un total de 5.215.104, según el reporte de la compilación proporcionado por Quartus II. Finalmente el mismo reporte, de la simulación y síntesis, indica que la frecuencia máxima de reloj a la que puede trabajar el módulo es de 37,15 MHz, con el peor tiempo t_{su}=16,44 ns y el peor tiempo t_{co}=10,85 ns.

6.2.3. Simulación del módulo `Laplaciana`

La función del módulo `Laplaciana` consiste en efectuar el promedio (*average*) de los píxeles vecinos al píxel a procesar. Adicionalmente cada píxel tiene un peso según la posición que tenga respecto del píxel a procesar. Esta etapa es en si la que realiza la última restricción necesaria para el cálculo del flujo óptico, llamada restricción de suavidad.

En el módulo `Laplaciana` se ingresan 8 píxeles por cada imagen para obtener así una componente de las dos requeridas, para el cálculo de la Laplaciana. Dicho de otra manera son necesarios dos circuitos similares para calcular las dos componentes de la Laplaciana (LU, LV), que representan la componente horizontal y vertical respectivamente, del vector de flujo óptico. El diseño de este módulo, explicado detalladamente en la sección 5.5, consiste básicamente en la suma de dos grupos de 4 píxeles de 9 bits con signo y de dos divisiones. Esas dos divisiones corresponden a cada grupo de sumas, una división es por 6 y la otra es por 12. Las dos divisiones pueden ser sustituidas por dos divisiones más sencillas, dos divisiones por 3, y adicionalmente dos desplazamientos uno de 1 bit y otro de 2 bits, estos desplazamientos equivalen a una división por 2 y otra división por 4 respectivamente. Este hecho hace que la división por 3 y el uso de desplazamientos, que son operaciones más sencillas, requiera menos recursos hardware y una latencia mucho menor para la ejecución de ambas operaciones.

Durante el proceso de la simulación, como se dijo en un principio, se busca optimizar los recursos hardware utilizados y mejorar la frecuencia de procesado. De tal manera que se procedió a realizar varias simulaciones con distintos tipos de estrategias de diseño. Una de ella fue realizar una división mediante una arquitectura *pipeline*. Esa estrategia aunque permitía ganar recursos hardware el ahorro era marginal, puesto que sólo se reducían 5 elementos lógicos, según el reporte de compilación&simulación. Por otro lado, en cuanto al tiempo de respuesta del circuito, se presentaba una latencia de 3 ciclos de reloj. Por estas razones se procedió a utilizar un circuito divisor puramente combinacional.

En la figura 6.6 se muestra la simulación de un circuito que realiza sólo el cálculo de una componente de la Laplaciana. Se puede observar las señales de entrada de los 8 píxeles *(a, b, c, d, e, f, g* y *h)* y la salida *Lapla*. La salida es la suma de los 8 píxeles con su respectiva ponderación. Los 8 vectores de prueba introducidos representan los 8 píxeles, de 9 bits cada uno, que serán procesados. La salida es del mismo tamaño que los píxeles de entrada, esto es que puede tomar valores de -256 a +255.

Debido a que la salida del módulo es registrada, los datos de salida son presentados un ciclo de reloj después de que se ingresan los datos de entrada. La señal *lapla_start*, que se habilita en estado alto, se utiliza para indicarle al módulo que los datos que se encuentran en la entrada deben ser procesados. A su vez el módulo da una respuesta a la siguiente etapa para que sepa en que momento se tienen los datos ya procesados, esto se hace mediante la señal *lapla_ack*. Si la señal *lapla_ack* está en estado alto indica que la salida es correspondiente a los datos que ingresaron para ser procesados, un ciclo de reloj antes. Estas dos señales son utilizadas por el módulo de control, que se vera más adelante, para manejar las diferentes etapas del sistema.

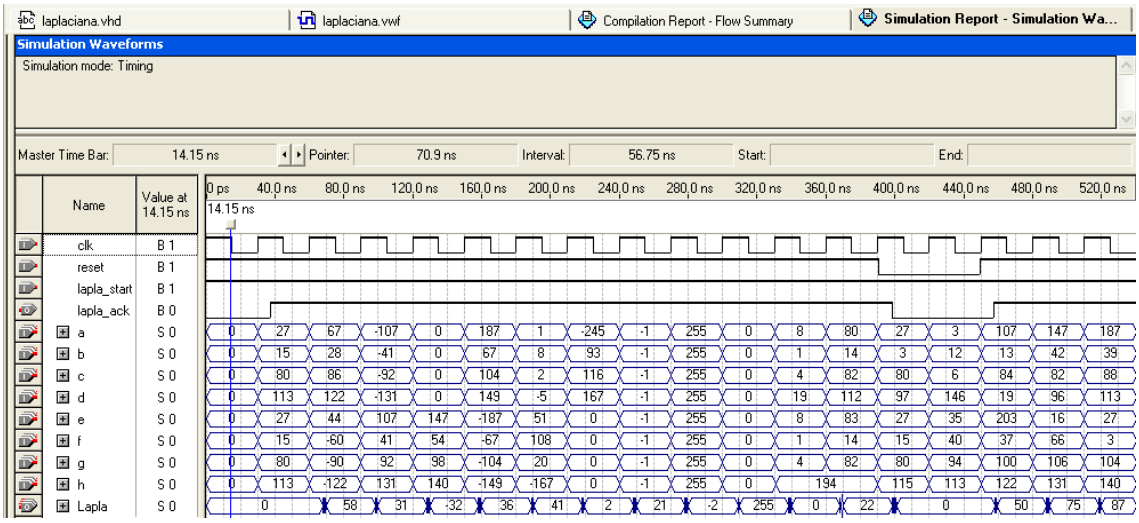


Figura 6.6: *Simulación del módulo* **`Laplaciana`**. *Este módulo obtiene sólo una componente.*

Finalmente, de la misma manera que en el caso de los módulos anteriormente simulados, se procede a realizar una comparación de los valores obtenidos mediante el software MATLAB y los obtenidos en la simulación, realizada con el software Quartus II. Esto se muestra en el cuadro 6.5.

Una vez que se comprueba que el circuito realiza la función para la cual fue diseñado, respetando las frecuencias de procesado requeridas por el sistema, se procede a evaluar el diseño del circuito para que sea capaz de procesar los píxeles necesarios para obtener 8 componentes de la Laplaciana simultáneamente. De esta manera se evalúa el diseño del circuito, explicado en la sección 5.5, capaz de procesar las 8 componentes de la Laplaciana. Por tal razón se procede a realizar las simulaciones funcional y digital (*timing simulation*), además de sintetizar el circuito

Entradas	**Salida**	
	MATLAB	**Quartus II**
a,b,c,d,e,f,g y h	*Lapla*	*Lapla*
0, 0, 0, 0, 0, 0, 0, 0	0	0
27, 15, 80, 113, 27, 15, 80, 113	58,75	58
67, 28, 86, 122, 44, -60, -90, -122	31,50	31
-107, -41, -92, -131, 107, 41, 92, 131	-30,91	-32
0, 0, 0, 0,147, 54, 98, 140	36,58	36
187, 67, 104, 149, -187, -67, -104, -149	42,25	41
1, 8, 2, -5, 51, 108, 20, -167	2	2
-245, 93, 116, 167, 0, 0, 0, 0	21,83	21
-1, -1, -1, -1, -1, -1, -1, -1	-1	-2
255, 255, 255, 255, 255, 255, 255, 255	255	255

Cuadro 6.5: *Valores calculados en coma flotante utilizando el sofware MATLAB y los valores obtenidos por Quartus II software de la arquitectura hardware diseñada que utiliza aritmética entera.*

correspondiente. En la figura 6.7 se muestra la simulación digital del diseño. Los vectores de prueba se encuentran en código octal, con la finalidad de facilitar la nomenclatura. La salida son 8 componentes de la Laplaciana. La señal de reset es activa en nivel bajo y el sistema funciona a una frecuencia de 33 MHz.

Debido a que la Laplaciana requiere dos componentes, una componente horizontal y otra componente la vertical (LU,LV) respectivamente, es necesario reproducir este mismo circuito dos veces para realizar la operación en paralelo y conseguir una arquitectura más eficiente.

Igual que en los dos módulos anteriores, adicionalmente a la simulación, se realizó la compilación, análisis y síntesis teniendo que se utilizan 1.408 elementos lógicos, de un total de 57.120, una frecuencia máxima de trabajo de 55,21 MHz, el peor tiempo t_{su}=24,395 ns y el peor tiempo t_{co}=10,176 ns. Todo esto según el reporte de la compilación proporcionado por Quartus II.

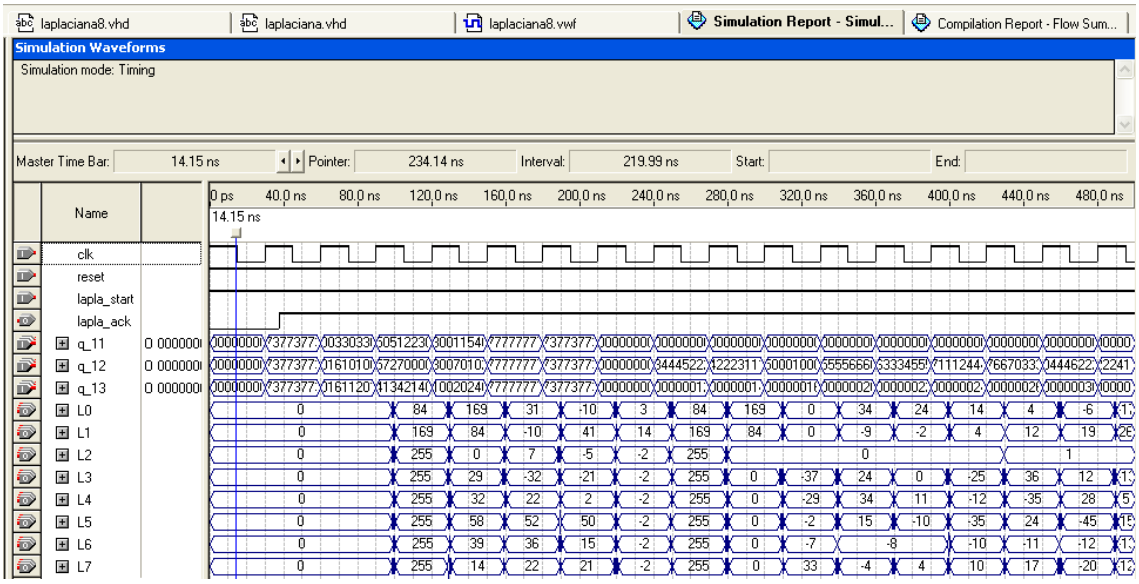


Figura 6.7: *Simulación del módulo* **`Laplaciana`**. *Aquí se obtienen ocho componentes simultáneamente.*

6.3. Simulación global

En esta simulación se evalúan todos los módulos diseñados y simulados de manera local. Además se incorporan las memorias FIFO's utilizadas en la arquitectura y los módulos de control necesarios, para llevar a cabo la sincronización de todos los bloques que conforman al sistema.

Es importante recordar lo expuesto en el capítulo 5, es decir, que la arquitectura aquí propuesta es del tipo flujo de datos. De tal manera que cada unidad procesadora o cada uno de los módulos diseñados cuenta con una memoria de entrada, para almacenar los datos a ser procesados, y con una memoria de salida, para almacenar los datos ya procesados. Además de eso cada etapa de procesamiento requiere un módulo de control, que a la vez está interconectado con otro módulo de control principal o de jerarquía superior.

En esta sección se abordará con más detalle los bloques de control y las componentes de almacenamiento utilizadas (memorias FIFO's). De esta manera aquí se detalla:

- Cómo se realiza el acceso a la memoria DDR SDRAM, localizada en la misma tarjeta de desarrollo, en la cual se implantará el sistema,
- Cómo se escribe en la memoria FIFO, utilizada para almacenar los datos a ser

procesados, localizada en la FPGA,

- Cómo se realiza la comunicación, sincronización y la transmisión de los datos entre los diferentes módulos y
- Cómo se escriben los resultados obtenidos (los vectores de flujo óptico) en la memoria DDR SDRAM.

Adicionalmente a esto, la simulación global sirve para realizar una explicación del funcionamiento de la arquitectura diseñada. Por lo tanto se aborda la explicación para crear, manejar y leer la tabla de cambios *LUT1*. La *LUT1* es generada por el módulo de detección de cambios y sirve para identificar los píxeles que cambiaron en el tiempo y que serán procesados.

Debido a que esta etapa de simulación es muy completa, además de compleja, se optó por realizar la simulación global de manera gradual e incremental. Primero se crea un proyecto, en el software Quartus II, que en este caso se le llamó `ctl_opt_flow`. Ya creado el proyecto se incluyen sólo algunos de los componentes que conformarán la arquitectura a diseñar, y se realiza una primera simulación. Una vez obtenidos los resultados esperados en la simulación, de esa parte del proyecto, se continúa creando e incorporando otra etapa o componentes necesarios para completar la arquitectura diseñada. Cada vez que se incorpora al proyecto una nueva etapa o módulo de procesamiento se realiza una nueva simulación, continuando con este proceso hasta concluir con la simulación global de la arquitectura.

Otro beneficio de esa estrategia de simulación es que permite ir explicando, de manera gradual e incremental, el funcionamiento de las diferentes etapas que conforman el sistema. Por todo esto la simulación global se llevo a cabo en tres etapas representativas, las cuales surgen de una división natural de la arquitectura, que son:

1. **Primera etapa:** Se realiza principalmente la simulación del módulo `LUT/Grad`, con los bloques previos y posteriores necesarios para su ejecución e interfaz con el siguiente módulo de procesamiento. Esto incluye la lectura de la memoria SDRAM, escritura de los píxeles en una memoria dentro de la FPGA, la posterior por el módulo `LUT/Grad`, que procesa y escribirá en una nueva memoria los datos calculados. También se considera el llenado de la tabla de cambios `LUT1`.
2. **Segunda etapa:** Aquí principalmente se lleva acabo la simulación del módulo `Velocidad` y los bloques previos y posteriores. Esta es la etapa más importante del sistema pues lee la `LUT1` y realiza la discriminación de las componentes de los píxeles que sí presentaron cambios significativos. Los píxeles seleccionados

para ser procesados son escritos en una nueva memoria, los cuales serán leídos y procesados por el módulo `Velocidad`. El resultado será escrito en una nueva unidad de almacenamiento.

3. **Tercera etapa:** En esta última etapa, la más laboriosa del sistema se lleva a cabo la simulación del módulo `Laplaciana`. Se simula el control de todos los bloques posteriores al módulo `Velocidad`. Entre estos bloques está una etapa que prepara la información para ser procesada por el módulo `Laplaciana`, memorias necesarias para almacenar el resultado y prepara la información para escribirla en la memoria SDRAM.

6.3.1. Primera etapa de la simulación global

Para llevar a cabo la primera etapa de la simulación global, en la cual se realiza la simulación de la lectura de la SDRAM y su posterior escritura en una memoria FIFO, localizada dentro de la FPGA, es necesario contar con cierta información específica e inicializar con valores establecidos. Para iniciar con la simulación del cálculo del flujo óptico es necesario contar con dos imágenes almacenadas previamente en la memoria SDRAM. La memoria SDRAM es de 256 MBytes y se encuentra localizada en la tarjeta de desarrollo Stratix PCI [Cor03b]. Debido a que la memoria DDR SDRAM es externa se requiere contar con el modelo de simulación de dicha memoria, el MT46V32M8 PLASTIC, 32MX8, disponible en *Micron Technology, Inc*[1].

Una vez que se cuenta con el modelo de simulación de la memoria DDR-SDRAM, es necesario crear un controlador para manejar dicha memoria. El controlador puede ser generado mediante el *SOPC Builder* o el *MegaWizard* de Altera y compilado y sintetizado mediante Quartus II software [Cor05a]. En este caso se utilizó el *MegaWizard*, que requiere un mayor conocimiento por parte del usuario de las características y parámetros del dispositivo a crear.

En este caso los parámetros utilizados, para crear el controlador de la memoria DDR SDRAM, son:

- Archivo de salida VHDL.
- *Presets*: Micron MT8VDDT3246HG-335C2.
- En la pestaña **Memory**, en el cuadro **Number of clock pairs from FPGA to memory** se indica 3.

[1] http://www.micron.com

- En la pestaña **Memory**, en el cuadro **Data bus width** indicar 32. Esto es debido a que el ancho del bus resultante sería de 64 bits, ya que la transferencia se realiza en cada flanco de reloj, esto es se transfieren 32 bits durante el flanco positivo y los otros 32 bits durante el flanco negativo.
- En la pestaña **Controller**, en el cuadro **Burst Length** se indica 8.
- En la pestaña **Board Timings**, en el cuadro **Memory DQ/DQS outputs to FRGA inputs, nominal delay** se indica 1200 ps.

Una vez creado el controlador de la memoria es posible iniciar con la generación de las memorias FIFO's necesarias.

La primera memoria a crear almacenará temporalmente 4 renglones de 256 Bytes (dos renglones de cada imagen) para proporcionar los píxeles al módulo `LUT/Grad`. El nombre asignado a esta memoria es `fifo_grad_in`. El nombre indica el tipo de memoria, a que módulo sirve y si almacenan datos de entrada o de salida de dicho módulo. En este caso es del tipo **(fifo)**, sirve al módulo **(grad)** y almacena datos que ingresarán al módulo **(in)**. La memoria es del tipo FIFO de doble puerto, con una entrada de reloj pero que maneja dos frecuencias distintas, una de 33 MHz y otra de 166 MHz. La memoria es creada con 4 módulos de 32 palabras cada uno, con un ancho de palabra de 8 píxeles de 8 bits (64 bits), lo que requiere un total de 8.192 bits.

Una vez procesados los píxeles por el módulo `LUT/Grad`, es necesario un dispositivo de almacenamiento temporal que mantendrá los datos de los gradientes y la tabla de los píxeles que cambiaron, resultantes del módulo. El nombre de las memorias que realizan este propósito son: `fifo_grad_out` y `LUT1`. La memoria `fifo_grad_out` es del tipo FIFO doble puerto. Su tamaño es de 32 palabras de 8 píxeles de 9 bits cada uno, para cada una de las tres componentes (I_x, I_y e I_t), lo que hace un total de 6.912 bits. La memoria tiene dos entradas de reloj, una de escritura a 33 MHz y otra de lectura a 166 MHz. La memoria `LUT1` es del tipo FIFO, también posee dos entradas de reloj, una de escritura a 33 MHz y otra de lectura a 166 MHz. La `LUT1` es de 32 palabras de 8 bits, pudiendo almacenar un total de 256 bits. Una vez almacenados los datos en las memorias `fifo_grad_out` y `LUT1`, los datos serán procesados por el módulo `Velocidad`. Todas las memorias aquí utilizadas fueron creadas por el *MegaWizard Plug-In Manager*, de Quartus II software, mediante la mega función *storage* (LPM_FIFO+).

Ahora bien, para gestionar la lectura de la memoria SDRAM, la escritura en la memoria `fifo_grad_in`, iniciar el cálculo de los gradientes y escribir estos gradientes

en la memoria `fifo_grad_out` es fundamental contar con un módulo de control. El módulo de control, denominado `Ctl_grad`, manejará y realizará todas las gestiones necesarias para llevar a cabo estos procesos.

El módulo de control `Ctl_grad` en realidad es una máquina de estados que en función de la información proporcionada por las memorias y los módulos procesadores, en este caso el `LUT/Grad`, deberá ejecutar comandos o activar señales para realizar una operación específica. Por ejemplo, si la memoria `fifo_grad_in` indica que está vacía, en ese momento el módulo de control realiza una petición de lectura al controlador de la memoria DDR SDRAM.

Ya creado el módulo de control, las memorias necesarias y contando con el modelo de simulación de la memoria SDRAM, es posible realizar la simulación funcional de esta parte del sistema. Para realizar la simulación se debe contar con dos imágenes almacenadas en la memoria SDRAM. Sin embargo, para efectos prácticos sólo se escribió una fracción de las imágenes, en particular fueron los 5 primeros renglones de las imágenes 8 y 9 de la figura *Rubic*, con el fin de agilizar el tiempo de simulación.

En las figuras 6.8 y 6.9 se muestran las señales que intervienen en esta primera etapa de la simulación global. En la figura 6.8 se muestran dos frecuencias de reloj: la *clk_33*=33 MHz y *clkddr*=166 MHz, que son utilizadas para generar una tercer señal de reloj (*clkfifowr*) utilizada por la memoria `fifo_grad_in`. El objeto de trabajar con dos frecuencias es para leer la memoria DDR SDRAM a una frecuencia de 166 MHz y además poder trabajar con la frecuencia de procesado, del módulo `LUT/Grad`. Esto es, leer la memoria `fifo_grad_in` y procesar sus datos a una frecuencia de 33 MHz. Con esta estrategia se gana un tiempo importante durante la lectura de los 131.072 píxeles de las dos imágenes, además de mantener un flujo de datos constante.

Las señales *local_addr, local_write_req, DdrAck, ddrrdreq_o*, *local_rdata_valid*, *Ddr Adr_o* y *RdDdrPix_o* son utilizadas para inicializar, direccionar y efectuar operaciones de lectura o escritura en la DDR SDRAM. La operación de lectura y escritura se realiza a 166 MHz, activándose las señales *wr11, wr12, wr21* y *wr22* e ingresando los datos *data11, data12, data21* y *data22* en la memoria. Una vez almacenados los 4 renglones es posible iniciar con el procesado de la información. Para esto se leen los 4 renglones simultáneamente, con la señal *rd_grad_in*, como se puede observar en la figura 6.9. También es posible observar, en la misma figura, que la lectura se realiza a una frecuencia de 33 MHz, a la misma frecuencia a la que se realiza el procesado de la información, como lo indica la señal *clkfifowr*. Por cada ciclo de reloj el módulo `LUT/Grad` obtiene los datos *Ix0* a *Ix7*, *Iy0* a *Iy7*, *It0* a *It7* y *ch0* a *ch7*. Estos datos

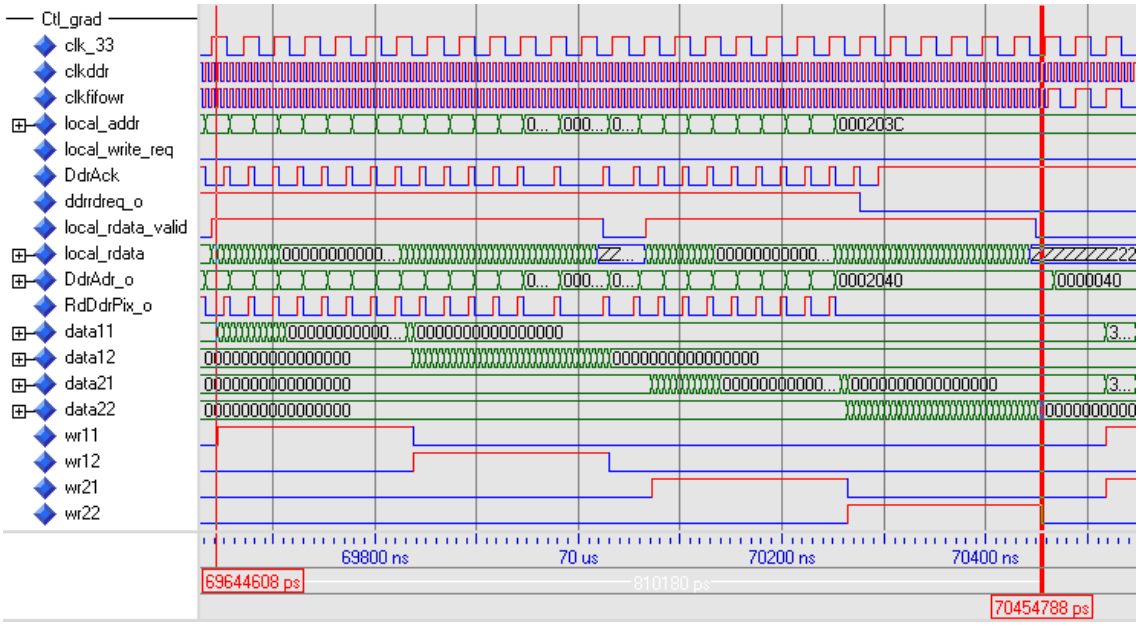


Figura 6.8: *Simulación global en la que se muestran las señales que intervienen en la etapa del procesado del módulo* `LUT/Grad`*, la conexión con la memoria que proporciona los datos de entrada y la memoria que almacena los datos de salida. También se muestran las señales necesarias para la lectura de la memoria DDR SDRAM.*

son escritos en la memoria de salida `fifo_grad_out` pero como dos bus de datos independientes; *data_grad_out* de 216 bits y *data_lut1* de 8 bits. Estos bus de datos son escritos cuando las señales *wr_grad_out* y *wr_lut1* se activan.

Simultáneamente a la lectura de la `fifo_grad_in` es posible observar que también se realiza una escritura en la misma memoria pero sólo de dos renglones, de los cuatro que la conforman, por lo tanto las señales *wr11* y *wr21* se muestran en alto.

Una vez que son procesados dos renglones, un renglón de cada imagen, se realiza una lectura a la memoria SDRAM y su correspondiente escritura en la memoria `fifo_grad_in` a una frecuencia de 166 MHz y se realiza nuevamente el procesamiento de la información, con el módulo `LUT/Grad`.

Las señales de control *full_grad_out* y *full_lut1* indican cuando las memorias `fifo_grad_out` y `LUT` respectivamente, están llenas. Estas señales le solicitan al módulo de control que detenga el proceso de cálculo de los gradientes y la tabla de detección de cambios. En consecuencia le módulo de control también detiene el proceso de lectura de la memoria `fifo_grad_in`. Por otro lado las señales *empty_grad_out*

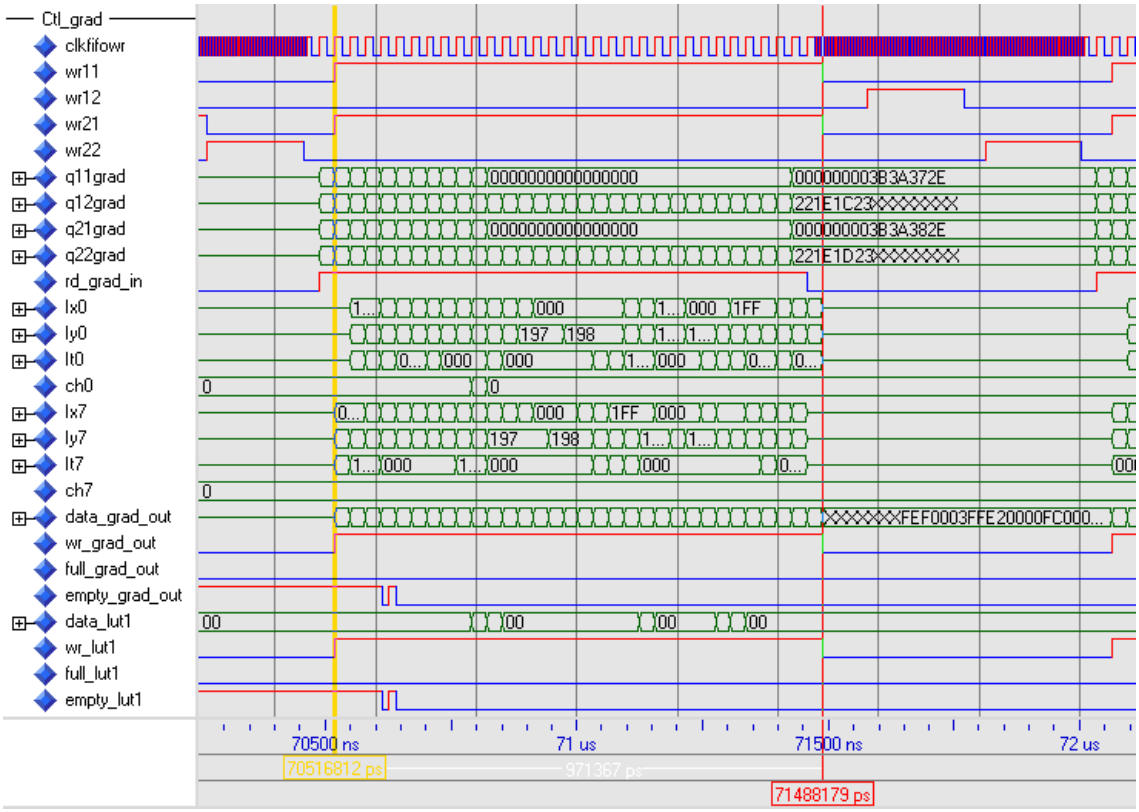


Figura 6.9: *Simulación global en la que se muestra las señales que intervienen en la etapa del procesado del módulo* ***LUT/Grad****, la conexión con la memoria que proporciona los datos de entrada y la memoria que almacena los datos de salida. También se muestran las señales necesarias para la lectura de la memoria DDR SDRAM.*

y *empty_lut1* indican al mismo módulo de control `Ctl_grad` que las memorias se encuentran vacías, para que reinicie el flujo de datos para ser procesado.

En resumen, aquí se realiza inicialmente una lectura de 1024 píxeles de la memoria DDR SDRAM. Los píxeles fueron escritos en la memoria `fifo_grad_in`, a una frecuencia de 166 MHz. Una vez concluida la escritura se realizó la lectura de la memoria `fifo_grad_in` y la escritura de los datos en el módulo `LUT/Grad` para ser procesados, a una frecuencia de 33 MHz. Un ciclo después, de la primera palabra escrita en el módulo `LUT/Grad`, se tienen datos ya procesados y son escritos en la memoria `fifo_grad_out` y en la tabla de cambios `LUT`. Ya concluido el procesado de los 1024 píxeles, realizado en 32 ciclos de reloj, se realiza una nueva lectura de la memoria DDR SDRAM. En este caso se leen sólo dos renglones (512 píxeles), un renglón de cada imagen, a una frecuencia de 166 MHz. En esta etapa de simulación

se crearon otros módulos necesarios para el diseño, de esta manera se realizó una simulación que integra los siguientes módulos:

- `LUT/Grad`, módulo para calcular los gradientes y de detección de cambios.
- `LUT1`, memoria con capacidad de almacenar 256 bits.
- `fifo_grad_in`, memoria con capacidad de almacenar 8.192 bits.
- `fifo_grad_out`, memoria con capacidad de almacenar 6.912 bits.
- `Ctl_grad`, módulo de control de esta etapa.
- `ddr_cntrl_top`, controlador de la memoria DDR SDRAM.
- `ddr_dim_model`, Modelo de simulación no sintetizable.

Todas las simulaciones aquí efectuadas y en lo sucesivo de esta sección han sido realizadas con el programa ModelSim-Altera, ver. 6.0E.

6.3.2. Segunda etapa de la simulación global

En esta etapa se realiza el proceso más importante de la técnica de procesado de imágenes guiado por cambios, técnica aquí propuesta y desarrollada. Este tipo de procesado se lleva a cabo en función de la tabla de detección de cambios, creada en la etapa anterior. Durante la lectura de la tabla `LUT1` se procede a discriminar las componentes de los gradientes que se obtuvieron. Las componentes que son discriminadas, o que no serán procesadas en esta etapa del proceso, son las calculadas de aquellos píxeles que no presentaron algún cambio significativo.

Esta segunda etapa de la simulación global se inicia con la lectura de las memorias `fifo_grad_out` y `LUT1`. La lectura se realiza a una frecuencia de 166 MHz y se efectúa al habilitar las señales *rd_grad_out* y *rd_lut1* mostradas en la figura 6.10. En función de los datos leídos *q_lut1*, de la tabla `LUT1`, y al momento de leer los datos de la memoria *q_grad_out* se lleva a cabo la discriminación de las componentes seleccionando aquellos que se van a utilizar. Mediante la señal *wr_vel_in* los datos *data_vel_in* son escritos y almacenados en una nueva memoria FIFO, llamada `fifo_vel_in`.

La memoria `fifo_vel_in` es creada con el **MegaWizard Plug-In Manager**, al igual que el resto de las memorias FIFO's aquí utilizadas. La memoria es de 256 renglones y un ancho de 45 bits. Es utilizada para almacenar las componentes I_x, I_y e I_t, además de las componentes LU y LV que serán calculadas después de la primer iteración. Inicialmente las componentes LU y LV tendrán un valor igual a cero.

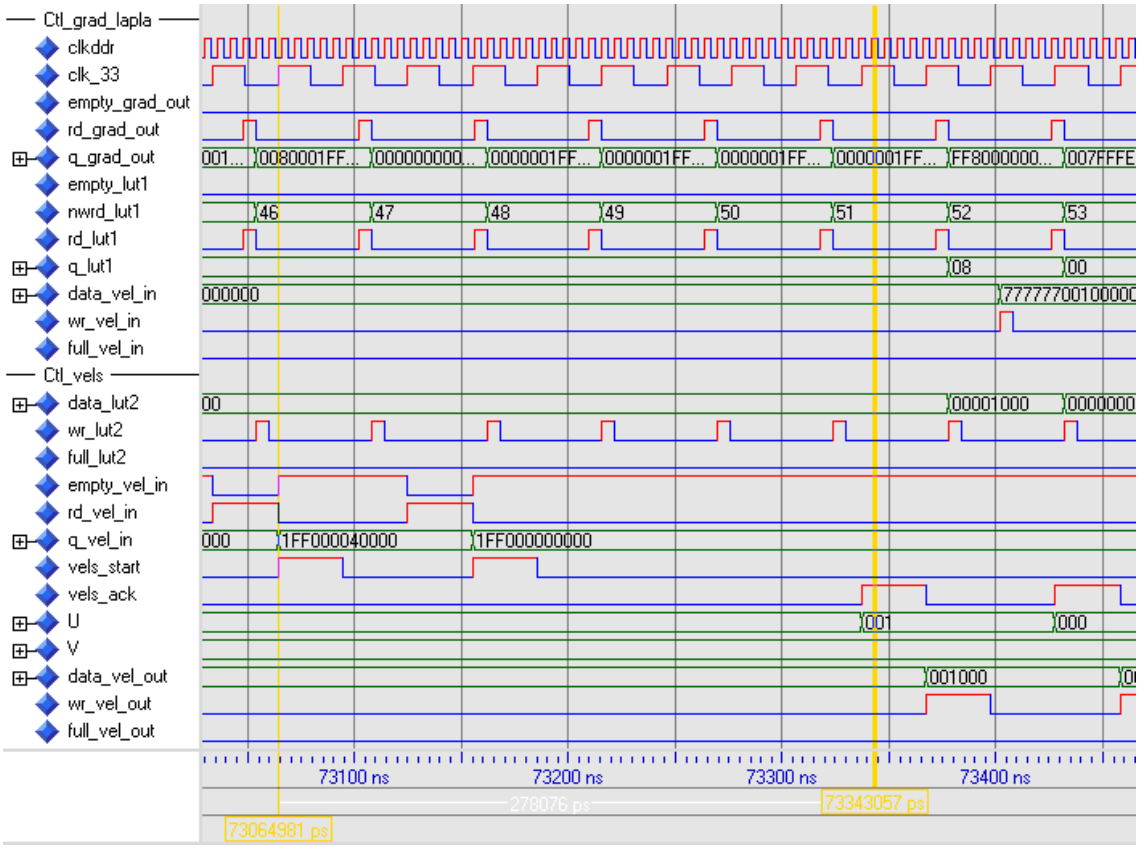


Figura 6.10: *Simulación global en la que se muestran las señales que intervienen en la etapa de cálculo de las velocidades, lectura de la* `LUT1`*, discriminación de los datos no utilizados y conexión entre las memorias que proporciona los datos de entrada y la memoria que almacena los datos de salida del módulo* `Velocidad`*.*

Por otro lado, los datos leídos *q_lut1* de la `LUT1`, una vez que hayan sido utilizados (después de 8 ciclos de reloj) son escritos como *data_lut2* en una nueva memoria `LUT2`, a una frecuencia de 166 MHz. La `LUT2` cuenta con una señal *full_lut2* que indica cuando se encuentra llena. De esta manera el módulo de control `Ctl_grad_lapla` detendrá el proceso de lectura de la memoria `LUT1`. El módulo de control `Ctl_vels` le indica al módulo de `Velocidad` en que momento debe de iniciar el procesado de la información mediante la señal *vels_start*. A su vez el módulo de velocidad indica cuando existe un dato valido (dato ya procesado), mediante la señal *vels_ack*, en su salida. Las salidas del módulo de velocidad están indicadas por las señales U y V.

Finalmente, las salidas del módulo `Velocidad` son escritas a una frecuencia de 33 MHz, en una nueva memoria `fifo_vel_out`. Esto será cuando la señal *wr_vel_out* sea habilitada por el módulo de control `Ctl_vels`. La memoria `fifo_vel_out` también

posee la señal *full_vel_out* para indicar que está llena y además posee dos entradas de reloj, una para escritura a 33 MHz y otra para lectura a 166 MHz. En esta etapa de simulación se crearon otros módulos necesarios para el diseño:

- `Velocidad`, módulo que calcula la velocidad.
- `fifo_vel_in`, memoria con capacidad de almacenar 11.520 bits.
- `fifo_vel_out`, memoria con capacidad de almacenar 4.608 bits.
- `LUT2`, memoria con capacidad de almacenar 16.384 bits.
- `Ctl_grad_lapla`, Discrimina las componentes, de los píxeles que no cambiaron y durante las iteraciones lee la memoria SDRAM.
- `Ctl_vels`, módulo de control de esta etapa.

Todas las simulaciones aquí efectuadas fueron hechas con el programa ModelSim-Altera, ver. 6.0E.

6.3.3. Tercera etapa de la simulación global

En esta última etapa de la simulación global también se incorporan nuevos elementos para llevar a cabo el procesado y almacenamiento temporal de la información, entre ellos están memorias FIFO's y un módulo de control que permite que los componentes realicen una transferencia y procesado de los datos de manera sincronizada. Esta etapa es, no por el tipo de las operaciones implementadas, sino por la necesidad de reconstruir las imágenes constituidas por las componentes LU y LV para para poder realizar la operación de la Laplaciana. La Laplaciana es en realidad la restricción de suavizado necesaria en la obtención del flujo óptico. La reconstrucción de las imágenes utiliza las componentes de los píxeles procesados, almacenados en la memoria `fifo_vel_out`, y los datos de la tabla de cambios `LUT2`.

Para llevar a cabo la reconstrucción de las dos imágenes que contienen las componentes de las velocidades y poder realizar el cálculo de la Laplaciana, es necesario leer primero la tabla `LUT2`. En la figura 6.11 se muestra el proceso de reconstrucción de las imágenes que conforman la velocidad o flujo óptico. Para leer la palabra *q_lut2* es necesario de al menos 8 ciclos de reloj y en cada ciclo se leerá un bit. Si el bit leído es 0 entonces el circuito de control `Ctl_vels` escribirá ceros en la entrada (*datau2* y *datav2*) de una nueva memoria FIFO, llamada `fifo_lapla_in`. En caso de que el bit leído sea 1, entonces se escribirán los datos de *q_vel_out*, leídos de la memoria `fifo_vel_out`, activando la señal *rd_vel_out*. Esto se puede observar con

más detalle en la figura 6.12. Obsérvese, por los cursores, que esta última figura es un zoom de la figura 6.11. En la figura 6.12 se muestra claramente que la palabra leída *q_lut2*=00000101, por lo tanto como existen dos 1's se deberán hacer dos lecturas mediante la señal *rd_vel_out*.

Al concluirse la lectura, la palabra *q_lut2* será escrita en una nueva memoria `LUT3` y se leerá una nueva palabra *q_lut2*. Este proceso se repite para todas las palabras de la tabla de cambios. Este proceso se realiza a una frecuencia de 166 MHz. La memoria `LUT3` pueda almacenar hasta 2.048 palabras de 8 bits.

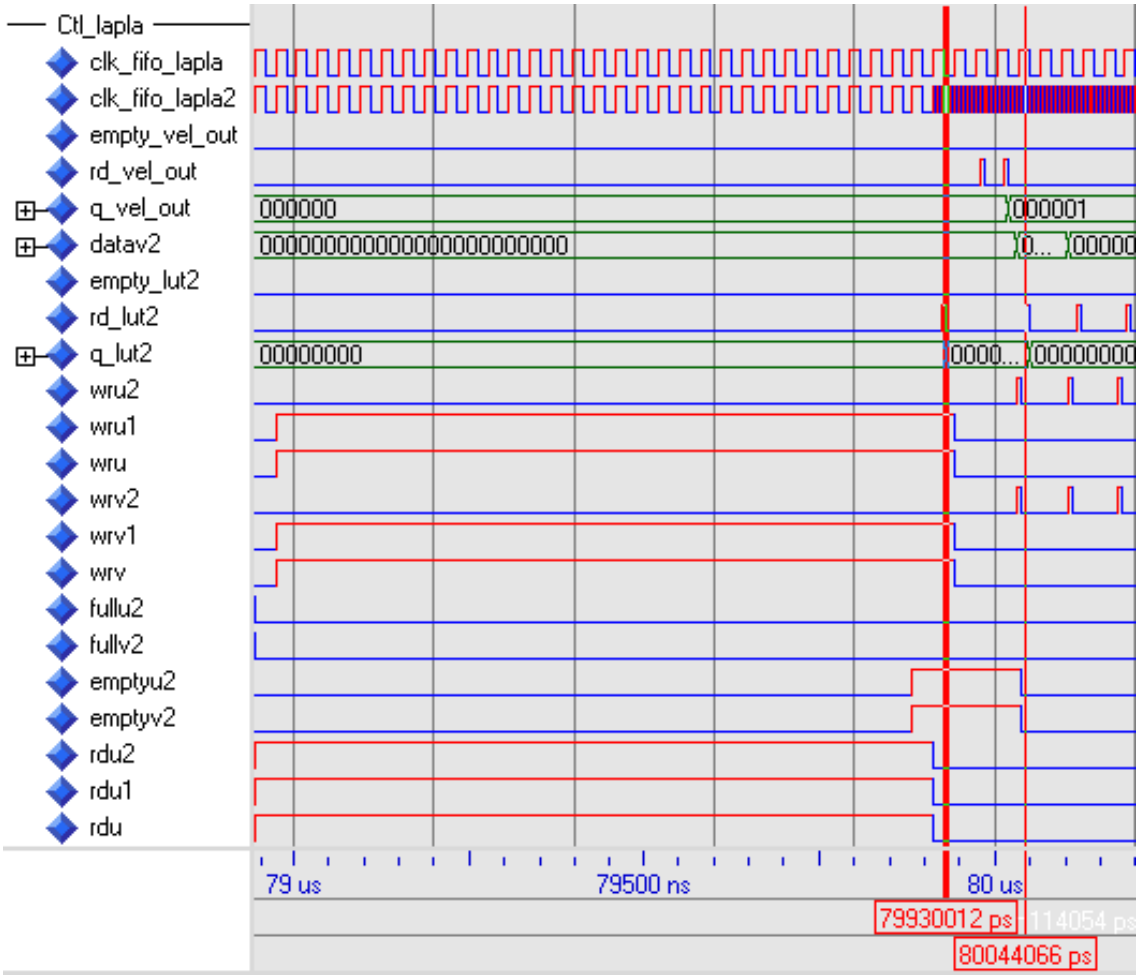


Figura 6.11: *Simulación global en la que se muestran las señales que intervienen en la etapa del procesado del módulo* ***`Laplaciana`****, la conexión con las memorias FIFO's de entrada y salida y la memoria que almacena la tabla de cambios.*

La memoria `fifo_lapla_in` almacena 3 renglones por cada imagen, de píxeles de 9 bits, por lo que su capacidad total de almacenamiento es de 13.824 bits. Posee

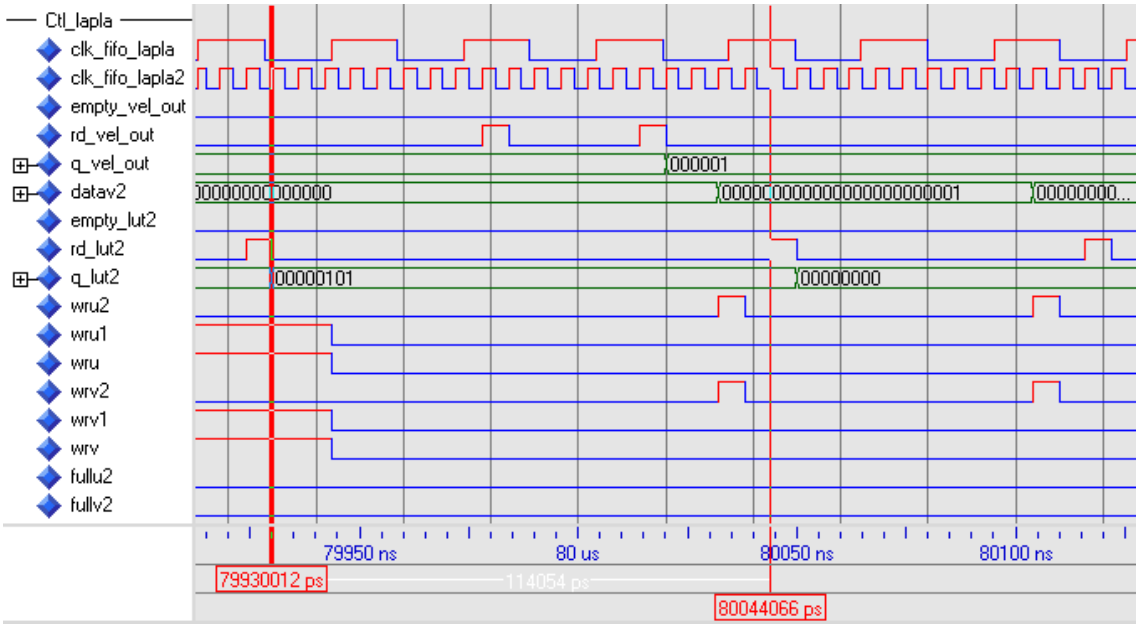


Figura 6.12: *Simulación global en la que se muestra la lectura de la* `LUT2` *para la reconstrucción de las imágenes para realizar el cálculo de la Laplaciana.*

dos entradas de reloj, una para escritura a 166 MHz y otra para lectura a 33 MHz, además cuenta con señales de lectura, escritura y para indicar si está vacía o llena. Estas señales son utilizadas por los módulos de control `Ctl_vels` y `Ctl_lapla`.

Cuando la memoria `fifo_lapla_in` indica que está llena al módulo `Ctl_lapla`, con las señales *fullu, fullu1* y *fullu2*, el módulo de control le informa al módulo `Lapla`, con la señal *lapla_start*, que inicie con el procesado de la información. De esta manera los datos *qu2, qu1, qu, qv2, qv1* y *qv* son leídos y procesados. Una vez obtenidas las componentes de salida *LU0 . . LU7* y *LV0 . . LV7*, el módulo de la laplaciana indica, por medio de las señales *lapla_acku* y *lapla_ackv*, que los datos ya están disponibles en la salida. Entonces los datos de salida *data_lapla_all* son escritos en la memoria `fifo_lapla_all` cuando la señal *wr_lapla_all* se activa. Este proceso es posible observarlo en la figura 6.13.

Una vez que se ha hecho el cálculo de la Laplaciana, es necesario preparar las componentes que serán utilizadas en la siguiente iteración, para el cálculo de las velocidades. De esta manera es necesario realizar nuevamente la discriminación de las componentes obtenidas, de los píxeles que no presentaron cambios significativos. Este proceso se realiza mediante una lectura de la tabla de cambios `LUT3`, siguiendo la misma estrategia cuando se leyó la tabla `LUT1`. Aquí en función del contenido se

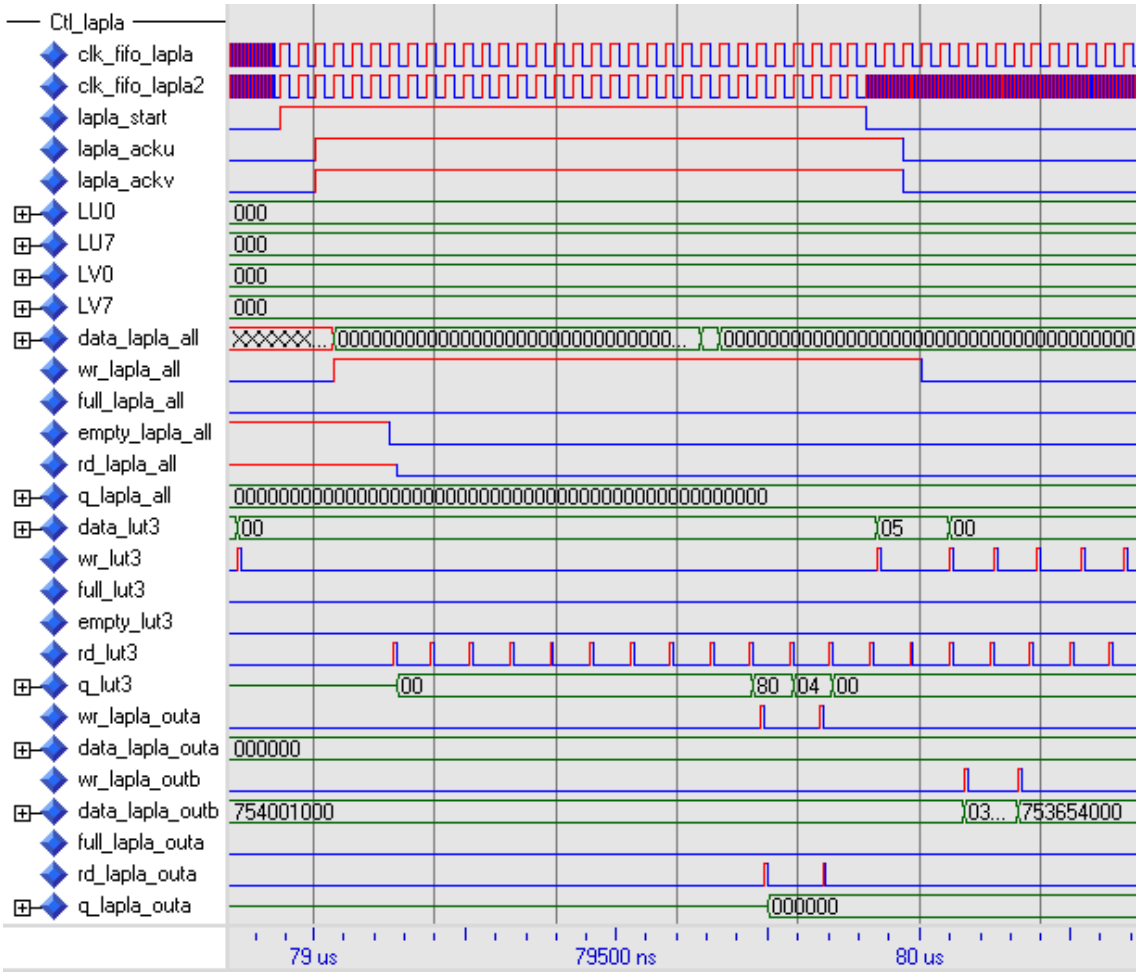


Figura 6.13: *Simulación global en la que se muestra la lectura de la* `LUT2` *para la reconstrucción de las imágenes para realizar el cálculo de la Laplaciana.*

leerá la memoria `fifo_lapla_all` y los datos de salida *q_lapla_all* son escritos en una nueva memoria FIFO, llamada `fifo_lapla_out`. Por otro lado cada palabra leída de la `LUT3` será almacenada en una tabla de cambios principal llamada `main_LUT`. Esta tabla de cambios principal, capaz de almacenar 4.096 palabras de 8 bits, es manejada por el módulo de control `Ctl_grad_lapla` y utilizada para las futuras iteraciones.

La memoria `fifo_lapla_out` en realidad está conformada por dos memorias que almacenan todas las componentes necesarias para realizar el cálculo de la velocidad. Las componentes almacenadas son: LU, LV, I_x, I_y e I_t, cada una de un tamaño de 9 bits. De esta manera una memoria almacena las componentes de la Laplaciana `fifo_lapla_outa` y la otra almacena las componentes de los gradientes `fifo_lapla_outb`. Ambas memorias trabajan a una frecuencia de lectura de 166

MHz. La diferencia entre estas dos memorias, es que la memoria `fifo_lapla_outa`, puede almacenar 65536 palabras de 18 bits, que corresponden a las componentes *LU*'s y *LV*'s, Mientras que la memoria `fifo_lapla_outb`, puede almacenar 65536 palabras de 27 bits, correspondientes a los gradientes. Cuando los datos ya se encuentran en la memoria `fifo_lapla_out` y están listos para ser leídos se puede decir que se concluyó la primera etapa o primera iteración para el cálculo del flujo óptico.

Para llevar a cabo una segunda iteración el módulo de control `Ctl_grad_lapla` se encarga de leer la memoria `fifo_lapla_out`. Las iteraciones se detendrán cuando el módulo de control `Ctl_vels` haya contado como hechas la cantidad de iteraciones indicadas a realizar desde un inicio. El resultado de las velocidades obtenidas serán las existentes en la memoria `fifo_vel_out` y serán escritas en la memoria DDR SDRAM, la escritura será controlada por el módulo `Ctl_vels`.

En esta etapa de simulación se crearon otros módulos necesarios para el diseño que fueron integrados al proyecto `ctl_opt_flow`. Los módulos integrados son:

- `Laplaciana`.
- `fifo_lapla_in`, memoria con capacidad de almacenar 13.824 bits.
- `fifo_lapla_all`, memoria con capacidad de almacenar 18.432 bits.
- `fifo_lapla_outa`, memoria con capacidad de almacenar 1.179.648 bits.
- `fifo_lapla_outb`, memoria con capacidad de almacenar 1.769.472 bits.
- `LUT3`, memoria con capacidad de almacenar 16.384 bits.
- `main_LUT`, memoria con capacidad de almacenar 32.768 bits.
- `Ctl_lapla`.

Es posible concluir esta sección con el cuadro 6.6 y 6.7. En los cuadros se especifican las características de todos los módulos que componen la simulación global.

6.4. Simulación con un banco de pruebas

Para llevar a cabo la simulación final es necesario integrar un nuevo bloque al diseño. Este nuevo bloque realiza una función necesaria para la comunicación con el bus PCI del computador y es llamado `MegaCore PCI`. Debido a que el `MegaCore PCI` es necesario para poder realizar la simulación final y verificar el correcto funcionamiento del sistema diseñado, es necesario definir lo que se denomina un banco de pruebas o *testbench*. Muchas veces esta sección es la más costosa, en cuanto a

Fichero	Tipo	Descripción
`ctl_opt_flow`	Cabecera	Inicializa todos los módulos.
`fifo_grad_in`	Memoria	Almacena dos renglones de cada imagen (8.192 bits). Sus datos son procesados por `LUT/Grad`.
`LUT/Grad`	Procesador	Módulo que efectúa el cálculo de los gradientes y realiza el proceso de detección de cambios.
`LUT1`	Memoria	Tabla de cambios detectados, almacena 256 bits. Sus datos fueron generados por `LUT/Grad`.
`fifo_grad_out`	Memoria	Almacena los gradientes calculados por el módulo `LUT/Grad`. Tiene una capacidad de almacenamiento de 6.912 bits.
`Ctl_grad`	Control	Controla las lecturas y escrituras de las memorias, incluyendo la SDRAM, para el procesado del módulo `LUT/Grad`.
`ddr_cntrl_top`	Control	Maneja la lectura/escritura de la DDR SDRAM.
`ddr_dim_model`	Memoria	Modelo de simulación de la memoria DDR SDRAM. No es sintetizable.
`Ctl_grad_lapla`	Control	Realiza la discriminación de las componentes, de los píxeles que no cambiaron, y las escribe en la memoria `fifo_vel_in`. Durante las iteraciones lee la memoria SDRAM.
`fifo_vel_in`	Memoria	Almacena sólo las componentes que serán procesadas. Puede almacenar hasta 256 valores de las 5 componentes utilizadas para efectuar el cálculo de la velocidad, lo que da un total de 11.520 bits.
`Velocidad`	Procesador	Calcula los vectores de flujo óptico, conformados por la componente horizontal y vertical (U, V). Tiene una latencia de 8 ciclos de reloj.
`fifo_vel_out`	Memoria	Almacena las componentes de los vectores de flujo óptico (U, V). Puede almacenar 4.608 bits.
`LUT2`	Memoria	Almacena los datos leídos de la `LUT1`. Sus datos serán utilizados en la etapa de la Laplaciana. Su capacidad de almacenamiento es de 16.384 bits.

Cuadro 6.6: *Módulos que conforman el proyecto* `ctl_opt_flow`*, utilizado para realizar la simulación global.*

Fichero	Tipo	Descripción
`Ctl_vels`	Control	Maneja los datos almacenados en la `fifo_vel_in` que son procesados por el módulo `Velocidad`. Los resultados los escribe en la `fifo_vel_out`.
`Ctl_lapla`*	Control	Reconstruye las imágenes de las velocidades utilizando la `LUT2` y la `fifo_vel_out`. También maneja el proceso de lectura/escritura de las memorias relacionadas con el módulo de la Laplaciana.
`fifo_lapla_in`	Memoria	Almacena 3 renglones de cada una de las dos imágenes de la velocidad, preparando así los datos para ser procesados por el módulo `Laplaciana`. Su capacidad es de 13.824 bits.
`Laplaciana`	Procesador	Calcula 8 pares de Laplacianas por cada ciclo de reloj. Su latencia es de dos ciclos de reloj.
`fifo_lapla_all`	Memoria	Almacena los resultados del módulo `Laplaciana`. Su capacidad es de 18.432 bits.
`LUT3`	Memoria	Almacena los datos leídos de la `LUT2`. Su capacidad es de 16.384 bits.
`Ctl_lapla`*	Control	Realiza la discriminación de los datos almacenados en la `fifo_lapla_all` utilizando la `LUT3`. Los datos son escritos en la `fifo_lapla_outa`.
`main_LUT`	Memoria	Almacena los datos leídos de la `LUT3`. Su capacidad es de 32.768 bits. Sus datos son utilizados en las sucesivas iteraciones y al final del cálculo del flujo óptico, después de las n iteraciones.
`fifo_lapla_outa`	Memoria	Almacena las componentes de las Laplacianas que serán utilizadas en las iteraciones siguientes. Su capacidad es de 1.179.648 bits.
`fifo_lapla_outb`	Memoria	Almacena las componentes de los gradientes que serán utilizados en las iteraciones siguientes. Su capacidad es de 1.769.472 bits.

Cuadro 6.7: *Módulos que conforman el proyecto* `ctl_opt_flow`*, utilizado para realizar la simulación global (continuación).*

tiempo se refiere, en el diseño de sistemas digitales.

La manera más directa de verificar el correcto funcionamiento de un modelo VHDL consistiría en cambiar las entradas y observar cómo evolucionan las salidas. Esta técnica resultaría muy útil para diseños sencillos, de hecho fue utilizada en este trabajo en la etapa de la simulación local, sin embargo, para modelos más complejos es mejor definir un banco de pruebas.

Un banco de pruebas no es más que la definición de un subconjunto de entradas con las que se va a comprobar el funcionamiento del circuito o modelo VHDL. Con el lenguaje de descripción de hardware VHDL es posible modelar un banco de pruebas independiente de la herramienta de simulación, en este caso se utiliza ModelSim-Altera para realizar la simulación. El código VHDL del banco de pruebas es utilizado sólo para llevar a cabo la comprobación funcional y nunca será sintetizable. Asimismo es necesario mantener separados los módulos que pertenezcan al banco de prueba respecto de los módulos que pertenecen al sistema diseñado y que será sintetizado.

La arquitectura del banco de pruebas, de tipo estructural, tiene como señales internas las entradas y salidas del circuito diseñado. El único componente es el correspondiente a la entidad que se desea simular. Si posteriormente se sintetiza el circuito diseñado, el mismo banco de pruebas utilizado para verificar la descripción pre-síntesis puede ser utilizado para simular el modelo VHDL post-síntesis generado por la herramienta software utilizada.

6.4.1. *MegaCore PCI*

Como se dijo anteriormente, es necesario incorporar un bloque que servirá de interfaz con el bus PCI del computador. El software de *Altera PCI Compiler* proporciona dos opciones para crear la interfaz PCI a la medida de las necesidades del diseño [Cor05b]:

1. **PCI Compiler con el MegaWizard.** Esta opción minimiza el tiempo de latencia, sin embargo, se debe contar con un mayor conocimiento del bus PCI, puesto que es un diseño a más bajo nivel.
2. **PCI Compiler con el SOPC Builder.** Esta opción es más rápida y sencilla de utilizar, crea un diseño simple e integrado automáticamente por el compilador. Su desventaja es que el tiempo de latencia, del sistema diseñado con esta herramienta, no es óptimo.

En este caso el módulo fue diseñado con el **MegaWizard Plug-In Manager**, utilizando Quartus II software v.5.1.0 y el compilador *PCI compiler v4.1.0* [Cor05b]. Para esto es necesario especificar la mega función PCI a utilizar, los parámetros internos, los archivos de diseño a generar e integrar manualmente las especificaciones de la mega función PCI dentro del *MegaWizard*. Con esto se tiene un mayor control de los módulo individuales que integran al la mega función PCI.

Para crear la interfaz PCI es necesario crear primero un proyecto en el software Quartus II, en este caso se creó el proyecto `pci_top`. Dentro del proyecto se ejecuta el *MegaWizard Plug-In Manager* y se selecciona la interfaz PCI *(PCI Compiler v.4.1.0)* del conjunto de Mega-Funciones. También se indica el tipo de archivo de salida a generar, en este caso fue VHDL, y se especifica la ruta donde será creado el archivo. Después de esto aparece una ventana en la cual se muestran varias opciones, como la información del Mega Core y los pasos necesarios para crear la función PCI. Básicamente para crear la función son necesarios 3 pasos.

1. Parameterize. Es el paso más importante y consiste en proporcionar los parámetros y características de la interfaz que se pretende diseñar.
2. Set Up Simulation. Aquí se indica si se quiere generar los archivos necesarios para llevar a cabo la simulación de la interfaz a diseñar. De esta manera se indica que se genere un modelo de simulación.
3. Generate. Es el paso final y sólo confirma la generación de los ficheros mediante un reporte de los archivos creados.

Sin lugar a duda el primer paso es el principal, pues allí se especifican los parámetros y características de la función que se quiere diseñar. En este caso se indica que la función PCI a crear es de 32 bits Master/Target (**pci_mt32**). También se especifican los valores de los registros de configuración y de los registros de dirección base (BAR's). En otras palabras, aquí se especifican todos los parámetros del espacio de configuración del dispositivo PCI a diseñar. Una descripción del bus PCI y de su espacio de configuración se agrega en el apéndice B.

En este caso los parámetros introducidos fueron:

- *Device ID* = 0x0006.
- *Vendor ID* = 0x1172.
- *Revision ID* = 0x21.
- *Subsystem ID* = 0xB102.

- *Subsys Vendor ID* = 0x1172.
- *BAR0* = 1 MByte de espacio de memoria.
- *BAR1* = 128 MByte de espacio de memoria *prefetchable*.

El registro BAR 0 siempre está habilitado y reserva 1 MByte de espacio de memoria. Sin embargo, es posible cambiar el registro para reservar un espacio de entrada/salida (I/O) o de memoria *prefetchable*. La razón de que el BAR 0 reserve 1 MByte, de espacio de memoria, es debido a que mapea todos los registros locales de lectura/escritura y los buffers FIFO de DMA localizados en el módulo `backend`. De hecho el espacio reservado es dividido en dos regiones de 512 KBytes: de 00000H - 7FFFFH es utilizada por los registros de configuración de la DDR SDRAM. De 80000H - FFFFFH es utilizada por los buffers FIFO para el DMA.

El registro BAR 1 reserva 128 MByte los cuales son mapeados para el módulo de la memoria DDR SDRAM. La memoria con la que se cuenta en la tarjeta de desarrollo es de 256 MByte.

Los rangos de direcciones de los registros BAR's son comparados con la dirección del bus PCI, por la función **pci_mt32**. Este hecho sucede durante la fase de direccionamiento y cuando se solicita, por ejemplo, una transacción Target del PCI. Si la dirección decodificada coincide con el espacio de configuración reservado para los registros BAR 0 y BAR 1, la función **pci_mt32** confirma con la señal *devseln* al bus PCI para aceptar la transacción. Después de esto la función **pci_mt32** activa las señales *lt_framen* y *lt_tsr[11..0]*, dentro del módulo `backend`, al control lógico Target notificando que ha sido solicitada una transacción Target. El control lógico Target (`targ_cntrl`) utiliza las señales *l_adro* y *lt_tsr[1..0]*, de la función **pci_mt32**, para determinar si la transacción es para la DDR SDRAM, los registros DMA o los buffer FIFO DMA. También es decodifica la señal *l_cmdo[3..0]* para determinar si es escritura o lectura a memoria.

En la figura 6.14, se muestra el diagrama funcional a bloques de la función **pci_mt32** y de todas las señales que intervienen en la interfaz.

6.4.2. Estructura del programa

Hasta este momento no se ha indicado a detalle sobre la organización del programa o la estructura del sistema aquí diseñado. En el capítulo 5 se habló de la metodología de diseño *Top-Down*, la cual es permitida por el lenguaje de descripción

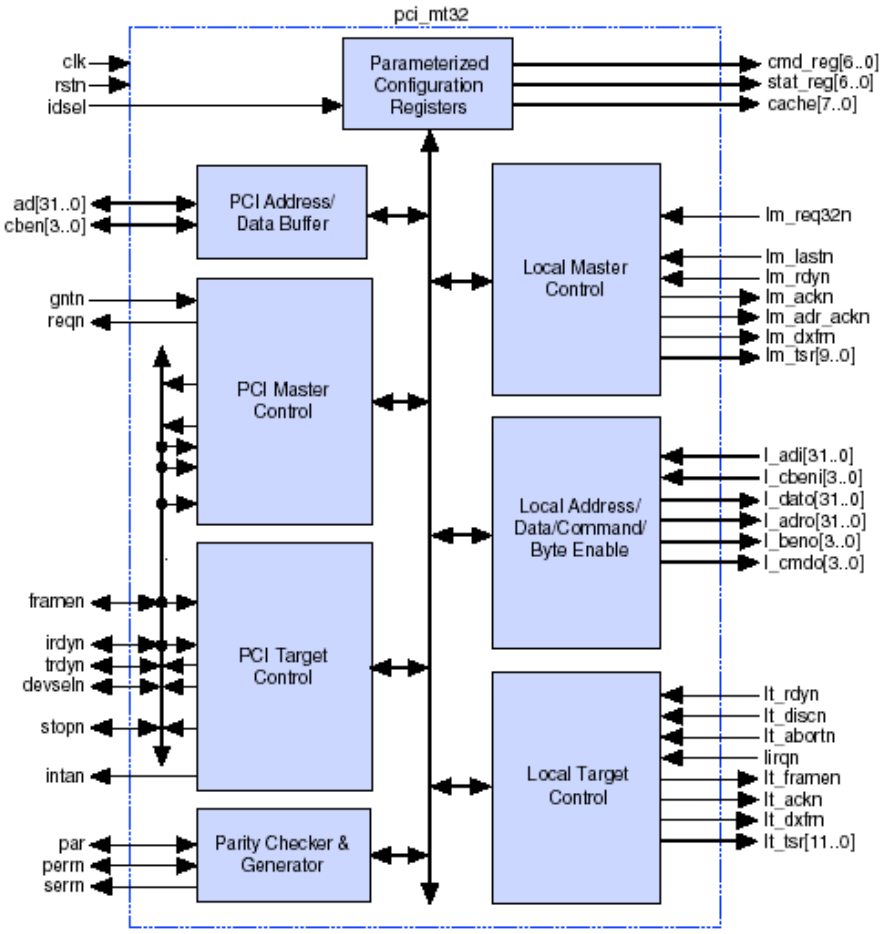


Figura 6.14: *Diagrama a bloques de la función* ***pci_mt32****, creada por el MegaWizard Plug-In Manager, mostrando las señales que la conforman.*

de hardware VHDL. Esta estrategia de diseño permite modelar el comportamiento de los bloques de alto nivel, simularlos y depurar la funcionalidad de alto nivel requerida antes de llegar a niveles más bajos de abstracción, de la implementación del diseño.

El lenguaje VHDL también permite utilizar los diferentes módulos creados en diferentes diseños y lo que es más importante permite la modularidad, muy útil cuando los diseños son muy complejos. La modularidad consiste en dividir o descomponer un diseño hardware y su descripción VHDL en unidades más pequeñas.

De esta manera el diseño del programa se realizó en forma modular permitiendo establecer una jerarquía como se muestra en la figura 6.15. Es posible observar que el módulo `stratix_top` es el archivo de nivel jerárquico superior del sistema diseñado.

De tal manera que la síntesis se realizará a partir de este módulo el cual incluye todos los archivos existentes y necesarios para que el sistema hardware funcione. Es decir, el fichero `stratix_top` contiene una instancia de cada uno de los submódulos utilizados, entre los cuales están: `pci_top, backend, ctl_opt_flow, ddr_intf, ddr_cntrl_top` y `ddr_pll_stratix`.

Por otro lado, cada uno de los submódulos puede estar dividido en otros componentes que poseerán un nivel jerárquico inferior, como se muestra en la figura 6.16, e incluso estos últimos también pueden estar conformados por otros de mucho menor nivel y así hasta llegar a los componentes más básicos diseñados en este trabajo. Un ejemplo de lo antes dicho es el caso del submódulo `ctl_opt_flow`, este submódulo está dividido en otros 5 bloques de un nivel jerárquico inferior. Uno de esos bloques, el `Ctl_grad`, se subdivide a su vez en otros 4 nuevos bloques y, aunque no está especificado en la figura, el bloque `grad8` contiene una división más que consiste de 8 entidades de un archivo básico llamado gradiente, de ahí su nombre gradiente8 o `grad8`.

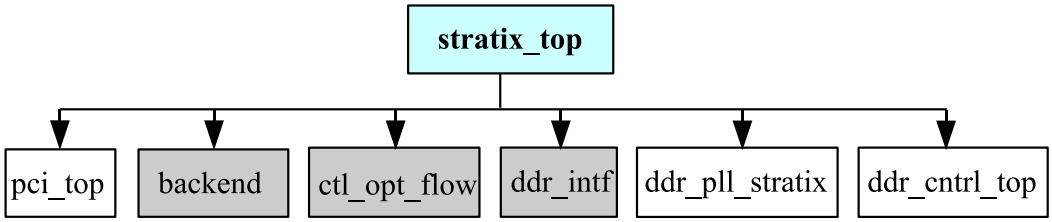


Figura 6.15: *Diagrama a bloques de la jerarquía superior de los programas que conforman el diseño hardware desarrollado.*

De la figura 6.15 el módulo `pci_top` implementa la interfaz PCI de 32 bits y es generada con el **MegaWizard Plug-In Manager** utilizando Quartus II software. El módulo `backend` representa el nivel superior de la interfaz PCI, pero del área local, que se conecta con el diseño implementado. En este módulo se implementan máquinas de estado para el control lógico target/master para lectura/escritura, también contiene memorias FIFO's y otras máquinas de estado, como la que realiza el proceso de acceso directo a memoria (DMA).

El módulo `backend` es proporcionado en el *kit profesional de la tarjeta de desarrollo Stratix PCI* pudiendo encontrar información detallada en [Cor02] [Cor05c]. El módulo está compuesto por otros submódulos, como se puede ver en la figura 6.16. De la misma manera el módulo `ddr_intf` es proporcionado por el *kit profesional de la tarjeta de desarrollo Stratix PCI*. Este módulo es el nivel superior de un

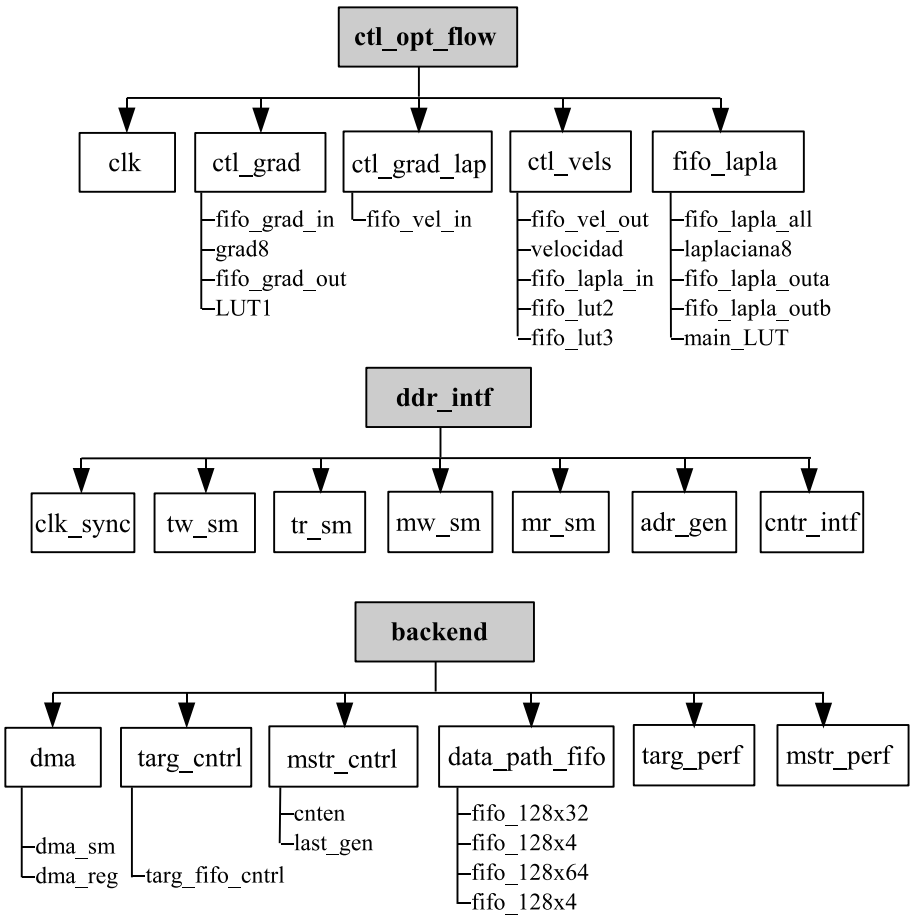


Figura 6.16: *Diagrama a bloques que muestra los programas que tienen una jerarquía inferior del diseño hardware desarrollado.*

grupo de ficheros que implementa la interfaz entre el bus PCI y la memoria DDR SDRAM. `tw_sm` implementa una máquina de estados que controla la escritura en modo target. `tr_sm` implementa una máquina de estados que controla la lectura en modo target. `mw_sm` implementa una máquina de estados que controla la escritura en modo master. `mr_sm` implementa una máquina de estados que controla la lectura en modo master. `adr_gen` genera la dirección de la memoria DDR SDRAM para la interfaz y `cntr_intf` toma las direcciones de la máquina de control de estados y entabla comunicación con el controlador de la DDR SDRAM localizado en el módulo `ddr_cntr_top`.

El módulo `ctl_opt_flow` representa el nivel más alto del grupo de bloques que implementan las funciones para el cálculo del flujo óptico. De la misma manera, se puede observar en la figura 6.16 que consta de 5 submódulos los cuales realizan varias

funciones como son: el submódulo `clk` genera una frecuencia de 33 MHz a partir de la frecuencia del bus PCI. El submódulo `ctl_grad` obtiene la tabla de cambios, los gradientes y realiza la lectura de la DDR SDRAM para escribir en las memorias FIFO's que se encuentran en este submódulo. El submódulo `ctl_grad_lap` lleva a cabo la lectura sólo de las componentes a procesar (que se obtienen de los píxeles que si cambiaron) y las escribe en una memoria FIFO que será utilizada por el submódulo `ctl_vels`. Este último submódulo realiza el cálculo de las velocidades y además implementa 4 memorias FIFO's, de las cuales dos de ellas son para almacenar la tabla de detección de cambios. Finalmente el submódulo `fifo_lapla` lleva a cabo el promediado de las velocidades (Laplaciana) y además implementa 4 memorias FIFO's.

El módulo `ddr_pll_stratix` es utilizado para generar la frecuencia de reloj para la DDR SDRAM. Finalmente el módulo `ddr_cntrl_top` es generado con el **MegaWizard Plug-In Manager**, utilizando Quartus II software v.5.1.0 y el compilador DDR SDRAM Controler v.3.3.0.

6.4.3. Simulación y síntesis

Para llevar acabo la simulación final y poder verificar la correcta funcionalidad del sistema diseñado es necesario definir un banco de pruebas o *testbench*. Un banco de pruebas no es más que la definición de un subconjunto de entradas con las que se va a comprobar el funcionamiento del circuito o modelo VHDL. El *kit profesional de la tarjeta de desarrollo Stratix PCI* incluye un banco de pruebas para la verificación de los sistemas implementados a base de sus *MegaCores* utilizados en la tarjeta de desarrollo. En este caso es para simular y verificar el funcionamiento de la función **pci_mt32**, representado por el módulo `pci_top`, y del controlador *PCI to DDR SDRAM*.

Con el uso de las herramientas que proporciona *Altera PCI testbench* y los modelos de simulación funcional IP de Altera, es posible realizar un prototipado "más rápido", puesto que facilitan en cierto grado la verificación del correcto funcionamiento de la aplicación desarrollada.

Para utilizar el *PCI testbench* se debe tener un conocimiento básico de la arquitectura y funcionamiento del bus PCI. La arquitectura y una breve explicación del bus PCI se puede encontrar en el apéndice B.

En la figura 6.17 se muestran todos los módulos proporcionados por el *PCI*

testbench y además el módulo diseñado para el procesado del flujo óptico guiado por cambios (`stratix_top`).

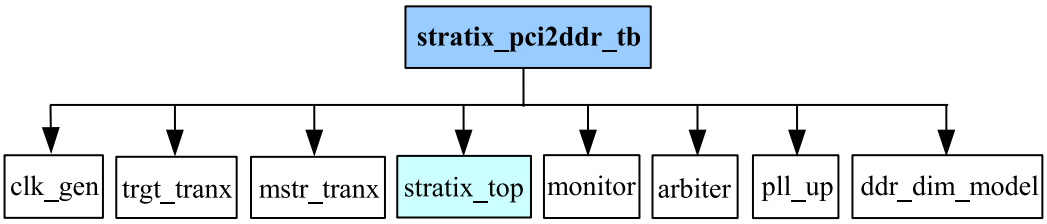


Figura 6.17: *Diagrama a bloques que muestra los módulos, contenidos en el* testbench *del proyecto de nivel superior.*

Durante la simulación con el banco de pruebas, fue necesario mantener separados los módulos que pertenecen al banco de pruebas de aquellos módulos o bloques que pertenecen al sistema diseñado, todos aquellos que están contenidos en `stratix_top`. Los módulos que pertenecen al banco de pruebas son necesarios sólo para realizar la simulación y los módulos o ficheros que pertenecen al sistema diseño, incluidos en el módulo `stratix_top`, serán sintetizado más adelante. La simulación se utilizó la herramienta de ModelSim-Altera v.6.0E y para el sintetizado el software Quartus II v.5.1.0.

La arquitectura del banco de pruebas, de tipo estructural, tiene como señales internas las entradas y salidas del circuito diseñado. El único componente es el correspondiente a la entidad que se desea simular. Si posteriormente se sintetiza el circuito diseñado, el mismo banco de pruebas utilizado para verificar la descripción pre-síntesis puede ser utilizado para simular el modelo VHDL post-síntesis generado por la herramienta software utilizada.

De esta manera una descripción del programa que inicializa el *PCI testbench*, los modelos de simulación funcional de los *MegaCore* utilizados y todas las conexiones necesarias entre estos componentes sería:

```
--(stratix_pci2ddr.vhd)

--**************************************
--Top-Level file of Altera PCI Testbench
--**************************************

ENTITY  stratix_pci2ddr  IS

END  stratix_pci2ddr;

ARCHITECTURE  behavior OF stratix_pci2ddr  IS
.
```

```
.
u0:mstr_tranx (..);
u1:trgt_tranx (..);
u2:monitor (..);
u3:arbiter (..);
u4:pull_up (..);
u5:clk_gen (..);
u6:ddr_dim_model (..);
u7:stratix_top (..); -- Modulo a sintetizar, incluye todo el sistema diseñado.
.
.
end behavior;
```

La simulación del sistema con el banco de pruebas, proporcionado por *Altera PCI testebench*, es modificado en su módulo `mstr_tranx`. La finalidad de modificar dicho módulo es para que lea dos imágenes, la 8 y 9 de la secuencia del *Rubic* que en principio se encuentran en la memoria principal del computador, y sean escritas en la memoria DDR SDRAM. En las figuras 6.18 y 6.19 se muestran las señales que intervienen en la transferencia de la información entre el bus PCI y la memoria DDR SDRAM, localizada en la tarjeta de desarrollo.

Para realizar la escritura de las dos imágenes en la memoria DDR SDRAM es necesario realizar una operación de escritura en modo *Target Write Transactions*. Para esto primero se debe de realizar una fase de direccionamiento mediante la activación de la señal *framen*, indicando la dirección en el bus *ad* y el tipo de operación a efectuar mediante el comando *cben*, que en este caso es de escritura *cben*= 0111, como se muestra con el primer cursor (de izquierda a derecha) de la figura 6.18.

Si la dirección es identificada por el dispositivo, en este caso por la función **pci_mt32** creada en la FPGA, entonces se comunica con el módulo `backend` activando la señal *lt_framen*, indicándole que hay una solicitud de escritura y con un valor en el bus *lt_tsr*. La función **pci_mt32** indica el comando y la dirección del bus, mediante las señales *l_cmdo y l_adro*. Al mismo tiempo activa las señales *devseln, trdyn, stopn y par*, pero en el siguiente ciclo de reloj.

Después de esto, el módulo `backend` indica que está listo para recibir los datos que provengan del la función **pci_mt32** activando la señal *lt_rdyn*. También la función **pci_mt32** le indica al bus PCI, mediante la activación de la señal *trdyn*, que está listo para aceptar datos. De esta manera los datos en del bus PCI (*ad*) serán manejados por la función **pci_mt32** utilizando la señal *l_dato*. Al mismo tiempo la función **pci_mt32** activa la señal *lt_ackn* indicando que el dato en el bus local *l_dato* es valido. Una vez que el dato o la ráfaga de datos ha sido hecha, dentro del módulo `backend`, la señal *lt_dxfrn* es activada y a su vez la señal *lt_tsr*, en uno de sus bits,

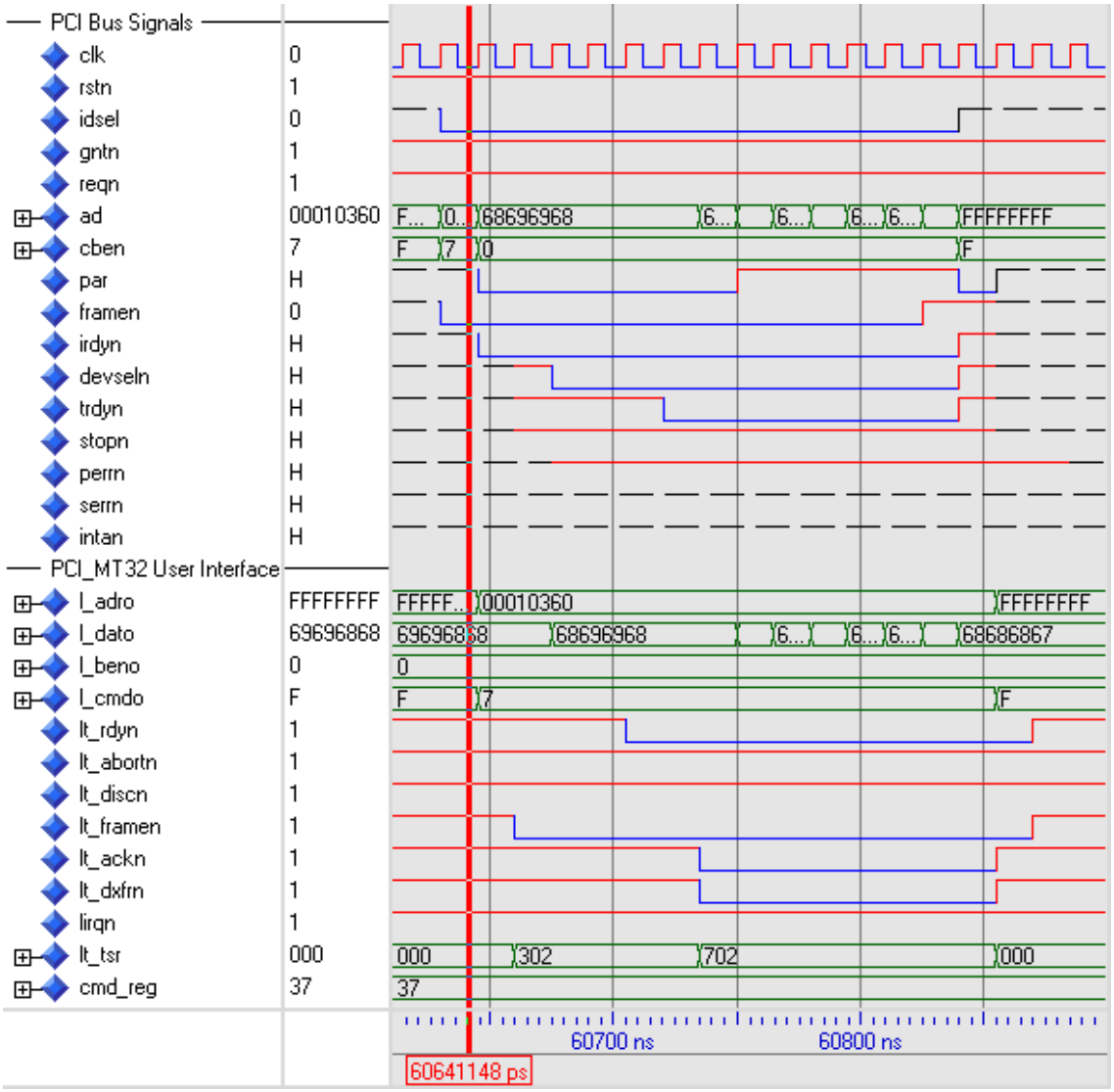


Figura 6.18: *Simulación donde se muestra la comunicación entre el bus PCI y la interfaz PCI diseñada.*

indicará que la transferencia de datos ha sido hecha satisfactoriamente.

La función **pci_mt32** también desactiva las señales *trdyn y devseln* al final de la transacción y después son puestas en alto y finalmente desactiva la señal *lt_framen* para indicar que no existen más datos en la memoria interna.

Dentro del módulo `backend` existe una memoria fifo `PCI-to-DDR FIFO` que almacena temporalmente los datos. El objetivo de esta memoria es almacenar temporalmente los datos para que el controlador de la interfaz pueda realizar la petición

de escritura al controlador de la memoria DDR SDRAM. La petición de escritura la hace mediante la señal *local_write_req*, mostrada en la figura 6.19 y referenciada con el segundo cursor (de izquierda a derecha) dentro del bloque de señales mostrado con la etiqueta *DDR Controller Signals*. Después de esto el controlador de la memoria contesta activando la señal *local_wdata_req* para que los datos estén listo en el bus *local_wdata* escribiendo 64 bits en cada ciclo de reloj.

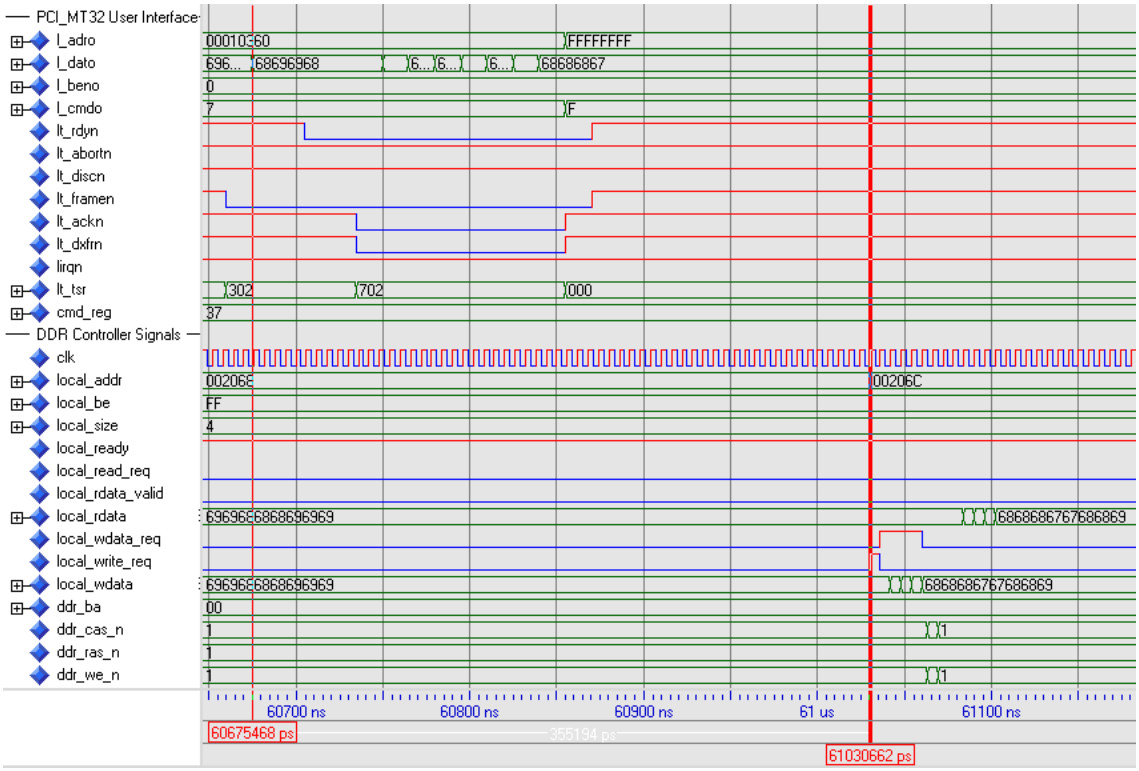


Figura 6.19: *Simulación donde se muestra la comunicación entre la interfaz PCI diseñada y la memoria DDR SDRAM.*

Una vez escritas las dos imágenes en la DDR SDRAM se indica al sistema, controlado por el submódulo `ctl_opt_flow`, que puede iniciar el procesado de los píxeles para el cálculo del flujo óptico guiado por cambios. Esto hace que el submódulo `ctl_opt_flow` genere una petición de lectura para la memoria DDR SDRAM y escriba los primeros 2 renglones de cada una de las dos imágenes en la memoria `fifo_grad_in`. Esta fase, de lectura en la DDR SDRAM - escritura en la `fifo_grad_in` se explicó en la etapa de la simulación global, específicamente en la sección de la primera etapa de la simulación global.

El resto de la simulación consiste en el procesamiento de las dos imágenes para

el cálculo del flujo óptico. Los tiempos de procesamiento en cada una de las etapas, en función de la simulación global, son los siguientes:

1. **Escritura PCI_a_DDRSDRAM.** En esta etapa de simulación se debe realizar una fase para inicializar la memoria SDRAM y poder llevar a cabo la escritura de los datos, desde el bus PCI a la memoria DDR SDRAM. El tiempo para realizar esta etapa de inicialización de la memoria y la escritura de las dos imágenes es de **2.882,56** μ**s**, según la simulación final.
2. **Lectura DDR SDRAM / escritura** `fifo_grad_in`**.** En esta segunda etapa se realiza la solicitud de lectura de la SDRAM y la escritura de la memoria `fifo_grad_in`, además se considera el tiempo necesario de procesado y almacenamiento de los datos en la memoria `fifo_grad_out` y en la `LUT1`. El tiempo de este proceso, según la simulación final, es de **384** μ**s** por par de imágenes procesadas, una vez que concluyó de escribirse las dos imágenes en la memoria de entrada `fifo_grad_in`. Es importante resaltar que este tiempo resulta debido a que los procesos se realizan de manera concurrente, la primera y esta segunda etapa.
3. **Discriminación de los píxeles.** Aquí se lleva a cabo la etapa de discriminación de los componentes que serán procesados debido a que los píxeles presentaron cambios significativos. Esta etapa inicia su procesamiento desde la primer palabra que se detecta en la `LUT1`. De este modo se lee la tabla de cambios `LUT1`, se lee la memoria `fifi_grad_out` y los datos que son seleccionados para ser procesados son escritos en la memoria `fifo_vel_in`. Este proceso se realiza en tan sólo **2** μ**s** debido a que se realiza de forma concurrente con las otras etapas y a que lo hace a una frecuencia de 166 MHz.
4. **Velocidad.** En esta etapa se procesan las componentes de los píxeles que sí cambiaron, es difícil saber en cuanto tiempo se realiza el procesado. La razón es porque depende directamente del número de cambios entre las dos imágenes consecutivas, por lo tanto también depende del umbral establecido en el sistema. En la simulación, para las dos imágenes del cubo de *Rubic* utilizadas en este caso se realizó en un tiempo de **920** μ**s**, con un umbral de t_h=3. Se puede agregar que en el peor de los casos, cuando ambas imágenes son totalmente diferentes, el tiempo aproximado sería de 1.920 μs para procesar 256×256 componentes a una frecuencia de 33 MHz.
5. **Laplaciana.** Esta etapa realiza el promedio de las velocidades vecinas, para esto es necesario leer de la memoria de salida del módulo de velocidad `fifo_vel_out`

y realizar una reconstrucción de las dos imágenes en una nueva memoria FIFO, cada una de las dos imágenes representa la componente de la velocidad horizontal y vertical respectivamente. Todo este proceso se realiza de forma concurrente, respecto de el resto de los módulos, por tal razón se hace en un tiempo de **3,42** μ**s**.

6. **Iteración y resultado.** Al igual que en la etapa de **Velocidad**, es difícil saber en cuanto tiempo se realiza el proceso de escribir en la SDRAM el resultado de las componentes de la Laplaciana, que implican sólo a los píxeles que presentaron cambios significativos. Esto es debido a que depende directamente del número de cambios entre las dos imágenes consecutivas y del umbral establecido en el sistema. En este caso el tiempo para este propósito es de **5.680**μ**s**. También es posible aproximar un tiempo para el peor de los casos, cuando ambas imágenes son totalmente diferentes, el tiempo aproximado sería de 11.530 μs para almacenar 5 componentes de 9 bits para 256×256 píxeles, trabajando a una frecuencia de 166 MHz y realizando 10 iteraciones.

Una vez hecha la verificación funcional del sistema implementado y que el resultado es adecuado y cae dentro de los valores esperados es posible llevar a cabo el sintetizado del circuito diseñado.

6.5. Síntesis Final

La síntesis del circuito `stratix_top`, que es el nivel superior del diseño, se realizó con Quartus II software v.5.1.0 utilizando la FPGA EP1S60F1020C6. Quartus II software genera un conjunto de ficheros en los que se reportan las características del proyecto compilado y sintetizado. Entre estos ficheros se encuentran:

1. Stratix_top.tan Timing Analyzer Summary.
2. Stratix_top.tan Timing Analyzer report.
3. Stratix_top.fit Fitter Status Summary.
4. Stratix_top.fit Fitter report.
5. Stratix_top.map Analysis & Synthesis Status Summary.
6. Stratix_top.map Analysis & Synthesis report.
7. Stratix_top.flow Flow report.
8. Stratix_top.asm Assembler report.

En el cuadro 6.8 sólo se despliega el reporte de compilación *Fitter Summary*.

Compilation Report		
Fitter Status	:	Successful - Mon Mar 12 16:23:00 2007
Quartus II Version	:	5.1 Build 176 10/26/2005 SJ Full Version
Revision Name	:	Stratix_top
Top-level Entity Name	:	Stratix_top
Family	:	Stratix
Device	:	EP1S60F1020C6
Timing Models	:	Final
Total logic elements	:	16,977 / 57,120 (30 %)
Total pins	:	250 / 782 (32 %)
Total virtual pins	:	0
Total memory bits	:	3,098,328 / 5,215,104 (59 %)
DSP block 9-bit elements	:	8 / 144 (6 %)
Total PLLs	:	2 / 12 (17 %)
Total DLLs	:	1 / 2 (50 %)

Cuadro 6.8: *Resultados obtenidos de la síntesis final utilizando la FPGA EP1S60F1020C6, de la familia Stratix de Altera.*

Parte III

Implementación y Experimentación

Capítulo 7

Implementación hardware

7.1. Introducción

Para llevar a cabo la implementación hardware es necesario conocer las características y funcionalidad de las herramientas utilizadas para el prototipado. También se debe tener una idea de la necesidad o necesidades que pudiese presentar el sistema a desarrollar durante la realización del proyecto.

Por lo tanto, es necesario conocer si no es perfectamente al menos estar al tanto de las capacidades del hardware utilizado para la implementación del sistema aquí desarrollado. En este caso es necesario conocer la tarjeta de desarrollo profesional Stratix PCI y las necesidades que presenta para su utilización.

7.1.1. La tarjeta de desarrollo Stratix PCI

La plataforma utilizada para la implementación de este sistema es la tarjeta de desarrollo profesional Stratix PCI de Altera. Esta tarjeta de desarrollo proporciona una plataforma muy completa para realizar pruebas y llevar a cabo la verificación de sistemas complejos. La tarjeta de desarrollo utiliza funciones IP y MegaCores de Altera, proporcionados a las universidades dentro de su programa universitario. Todo esto permite un ahorro de tiempo debido a que no es necesario realizar un trabajo adicional, que se encuentre fuera del diseño a implementar, permitiendo así dedicar un poco más de tiempo al desarrollo del sistema principal. En este caso el sistema a desarrollar es la propuesta de realizar un procesado de imágenes guiado por cambios aplicado al cálculo del flujo óptico.

Existen más componentes, junto con la tarjeta Stratix PCI, que se incluyen dentro del mismo *kit* de desarrollo. Los componentes más importantes, que integra el *kit* de desarrollo [Cor05c], son:

1. *Tarjeta de desarrollo profesional Stratix PCI.* Es una plataforma de desarrollo que es posible utilizar como un componente independiente o conectado al bus PCI del computador. Soporta una interfaz PCI de 3,3 V o 5 V, de 32 o 64 bits y frecuencias de 33 y 66 MHz.
2. *Diseño de referencia PCI a DDR.* Se agrega un diseño con el código abierto que puede ser modificado por el usuario. Este código realiza la comunicación entre el bus PCI y la memoria DDR SDRAM, que se encuentra en la tarjeta de desarrollo.
3. *Aplicación.* Es el código fuente para una aplicación de Windows que es posible utilizar para la configuración del dispositivo Stratix. Funciona únicamente cuando la tarjeta de desarrollo esta conectada en un bus PCI de 64 bits.
4. *Compilador para el MegaCore PCI.* Para la generación del MegaCore PCI es necesario el compilador que permite generar una función específica, ya sea 32 ó 64 bits y pueda trabajar como maestro o esclavo, para la interfaz PCI.
5. *Mega Función para generar el DDR SDRAM Controller.* Para la generación del MegaCore DDR SDRAM Controller es necesario contar con el módulo que permita generar el controlador de la memoria DDR SDRAM. La función resultante permite inicializar la memoria, solicitar lectura/escritura, etc.
6. *Software.* El software que incluye el *kit* es el Quartus II que permite realizar el análisis y síntesis del sistema a desarrollar. También se incluye un *shareware* para el diseño de controladores.
7. *Documentación* Se proporcionan los ficheros (**.vhd**) del diseño de referencia, del *testbench* y también manuales de los componentes del *kit* de desarrollo.

Los principales componentes en los cuales la tarjeta de desarrollo basa su flexibilidad son [Cor03b]:

- Dispositivo Principal: FPGA Stratix, EP1S60F1020C6.
- Memoria de 256 Mbytes. PC333 DDR SDRAM (SODIMM).
- Memoria Flash con capacidad de 64 Mbits.
- FPGA EPM3256ATC144, para configuración.
- Interfaz para programación USBBlaster II.

- Interfaz del Bus PCI, compatible con bus de 32 ó 64 bits.
- Interfaz Bel bus PCI compatible con 3,3 V o 5 V.
- Circuitos osciladores de 33 MHz y 100 MHz.

Además de los componentes antes citados existen otros que se pueden observar en la figura 7.1.

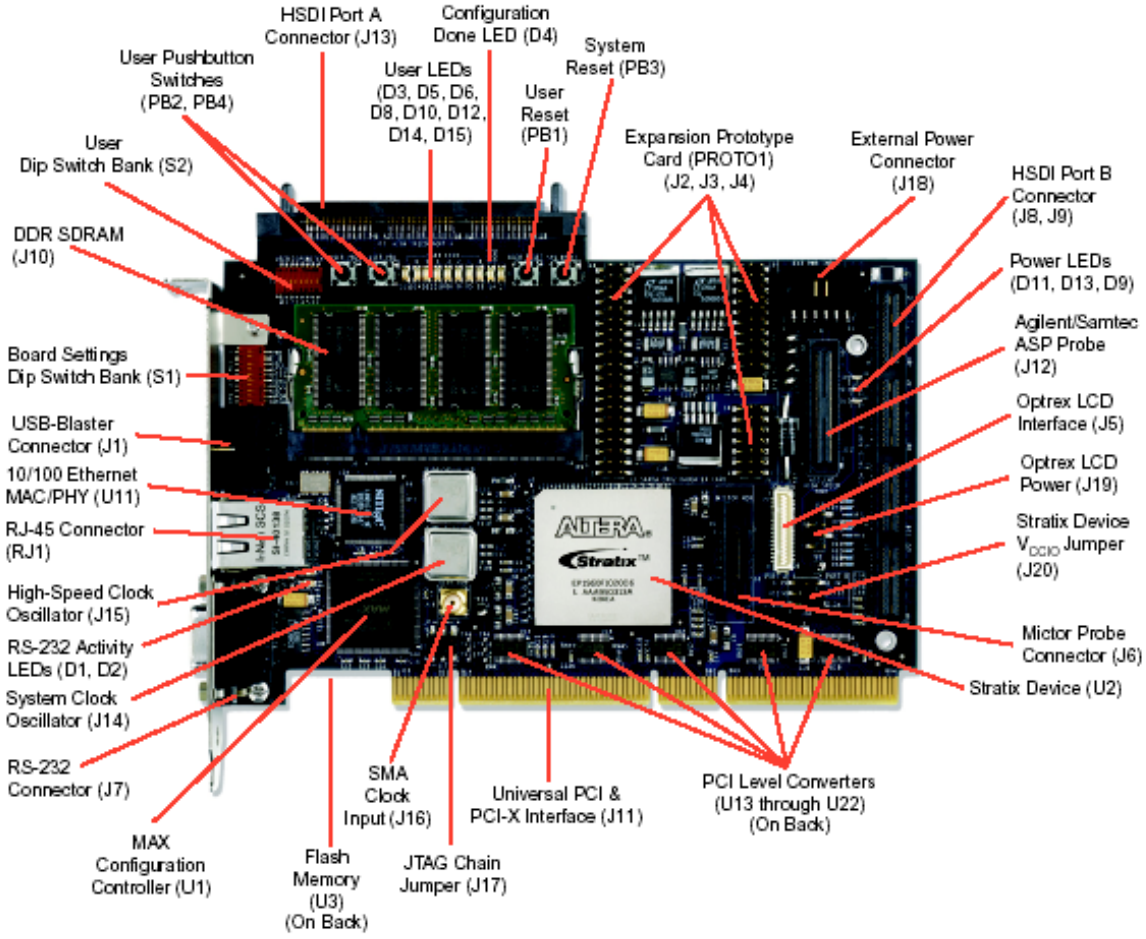


Figura 7.1: *Imagen de la tarjeta de desarrollo Stratix PCI, en la cual se indican los componentes que la conforman.*

Una vez estudiados y analizados los componentes y manuales [Cor05b], [Cor05c], [Cor04], [Cor05d], [Cor03b] y [Cor05a], para poder usar la tarjeta de desarrollo se reconoce que existen requisitos mínimos necesarios para llevar a cabo la implementación y prototipado del sistema en la plataforma de desarrollo Stratix PCI. Los requisitos se pueden dividir en dos tipos: software y hardware.

Requisitos del tipo software

Para utilizar la aplicación del *kit* de la tarjeta de desarrollo Stratix PCI es necesario contar con alguno de los siguientes sistemas operativos: Windows XP, Windows 2000 o Windows NT v. 4.0. SP 6. La razón es debido a que la aplicación proporcionada en el *kit* trabaja bajo ambiente Windows.

Además del sistema operativo, también es necesario contar con otros programas instalados en la PC, como son: Quartus II software, el compilador PCI, la Mega Función para generar el controlador, ModelSim-Altera y Jungo WinDriver. Quartus II software, que trabaja sobre Windows, sirve para crear un nuevo diseño, (sintetizar, simular, compilar, analizar y programar) para implementarlo en la tarjeta de desarrollo Stratix. El compilador PCI sirve para generar la interfaz PCI con características especiales, de acuerdo a las necesidades del proyecto o usuario. La Mega Función sirve para generar el controlador de la memoria DDR SDRAM, ModelSim-Altera es usado únicamente para realizar simulaciones complejas y la herramienta Jungo WinDriver es utilizado para crear los controladores, para que funcione la interfaz creada para el bus PCI.

Por otro lado, también son necesarios ciertos ficheros como son los *constraint files*. Los *constraint files* son proporcionados dentro del *kit* de desarrollo y específicamente indican la ubicación de las terminales, tipos de señales y tiempos de respuesta de las funciones PCI MegaCore y controlador DDR SDRAM MegaCore. Los *constraint files* son sumamente importantes e incorporan una restricción respecto a la versión de los programas utilizados, ya que son creados con un software específico, por tal motivo deben ser compilados y sintetizados, cuando se realicen nuevos proyectos, respetando el software con que fueron creados. Para este caso particular, donde los *constraint files* fueron creados para el dispositivo EP1S60F1020C6, se utilizó el compilador del PCI v.4.1.0, Quartus II software v.5.1.0. y la Mega Función utilizada para generar el controlador de la DDR SDRAM, fue la versión 3.3.0. Es necesario respetar las versiones de los programas para los que fueron creados los *constraint files*, ya que de lo contrario, es posible que se tengan violaciones de tiempo en las señales del bus PCI y de la DDR SDRAM. Otro inconveniente que puede surgir es que alguno de los programas no soporten el dispositivo utilizado.

Finalmente es necesario un software de desarrollo de aplicaciones, una vez que se cuente con el controlador que comunique el nuevo dispositivo conectado al bus PCI con el sistema operativo. Como se especificó se utilizó el Jungo WinDriver para crear el controlador necesario. El software que se utilizó para crear la interfaz visual

con el usuario fue C++ Builder de Borland.

Requisitos del tipo hardware

Los requisitos hardware para poder utilizar la plataforma de desarrollo Stratix PCI profesional en la computadora son básicamente un bus de 32 ó 64 bits y que trabaje a 3.3 V ó 5 V. Por tal motivo el computador deberá tener al menos dos *slots* PCI disponibles. Uno para la tarjeta de desarrollo Stratix PCI y otro bus de 32 bits para la cámara que captura las imágenes que serán procesadas.

Sin embargo, para configurar la FPGA (Stratix, EP1S60F1020C6) mediante la aplicación incluida en el *kit* de desarrollo es necesario que exista disponible un bus PCI de 64 bits. En otro caso se deberá programar el dispositivo mediante otro método [Cor05c]. El otro método para programar la FPGA es mediante la interfaz JTAG lo que hace necesario un puerto paralelo y por lo tanto el cable USBBlaster II para descargar el fichero.

El software utilizado para realizar nuevos proyectos hace que a su vez se requieran ciertas características específicas del computador con que se desea desarrollar el sistema. Por ejemplo, Quartus II software requiere los siguientes requisitos: Pentium 4 a 1 GHz o superior, 512 MB o más de memoria local y tener acceso a una memoria virtual de 2 GB o más. Así mismo para la instalación del software son necesarios varios componentes, que comúnmente se encuentran en un computador personal, como son: CD-ROM, DVD-ROM, monitor, etc.

7.2. Prototipado del diseño

El prototipado de un sistema sobre la plataforma de desarrollo *Stratix PCI development board* consiste básicamente en 5 pasos bien definidos:

1. Crear el diseño
2. Simular el diseño
3. Compilación y síntesis del diseño
4. Configuración de la FPGA
5. Verificación hardware del diseño

Los tres primeros pasos se realizaron en el capítulo 5 y 6. El diseño se culminó con la obtención del fichero `stratix_top.vhd` el cual incluye todos los módulos diseñados y listos para ser sintetizados. En la simulación se verifico el funcionamiento y los tiempos de respuesta del sistema. Dentro de esa etapa también se realizaron algunas mejoras.

Durante la compilación y síntesis del diseño, donde el proyecto a sintetizar fue el `stratix_top.vhd`, se obtuvieron los archivos necesarios para la programación de la FPGA. Cuando se realiza la compilación se deben especificar en *Device&Pin Options* los ficheros a crear, utilizados para la programación del dispositivo. En este caso los ficheros obtenidos fueron del tipo *Raw Binary File (**stratix_top.rbf**)* y del tipo *SRAM Object File (**stratix_top.sof**)*. De estos dos ficheros se utiliza uno u otro, según el método de programación. Antes de compilar y sintetizar el proyecto en Quartus II se deben de incorporar los *constraint files* mediante la consola TCL de Quartus II.

De esta manera sólo falta realizar los dos últimos pasos, la programación de la FPGA y realizar la verificación hardware del diseño.

7.3. Configuración de la FPGA

La tarjeta de desarrollo Stratix PCI soporta dos métodos de configuración, los cuales son [Cor05c]:

1. **Configuración mediante la memoria flash.** Este método de configuración se realiza utilizando la memoria flash localizada en la misma tarjeta de desarrollo y se utiliza el fichero **stratix_top.rbf**.
2. **Configuración mediante la interfaz JTAG.** Para este método se emplea el USB Blaster II download cable, que viene dentro del *kit* de desarrollo, y se utiliza el fichero **stratix_top.sof**.

Configuración mediante la memoria flash

El *kit* de desarrollo incluye una aplicación la cual utiliza algunos componentes que integran la tarjeta de desarrollo Stratix PCI. La aplicación es posible ejecutarla sólo si la tarjeta de desarrollo es conectada a un bus PCI de 64 bits. En este caso la tarjeta contiene una FPGA, la EPM3256AT144, y una memoria flash para la configuración de la tarjeta. El dispositivo EPM3256 implementa un circuito de control para la

Switch S1		Sección de la Memoria Flash
Posición 9	**Posición 10**	
Off	Off	0
Off	On	1
On	Off	2
On	On	3

Cuadro 7.1: *Posición de los* dipswitch *para la selección de una sección específica de la memoria flash*

programación de la Stratix utilizando los datos contenidos en la memoria flash. Este proceso tiene efecto una vez que ocurra alguno de los siguientes eventos:

- Cuando se enciende la tarjeta de desarrollo o cuando se enciende la computadora y la tarjeta esta conectada al bus PCI.
- Cuando se reinicia la computadora o la tarjeta de desarrollo, utilizando el botón de reset PB3.
- Cuando la aplicación incluida en el *kit* se utilice para configurar la tarjeta de desarrollo.

En el primer caso, al encenderse la tarjeta o la computadora, se configura por defecto la Stratix con el programa almacenado en la sección 0 de la memoria flash. La memoria flash contiene 4 secciones en las cuales se pueden almacenar 4 archivos de configuración distintos. Sin embargo, la sección 0 contiene el archivo de configuración de defecto hecho de fábrica que ejecuta la aplicación del *kit* de desarrollo y no es posible acceder a esa sección para escribir o cargar un fichero.

En el segundo caso, la aplicación contenida en el *kit* de desarrollo, permite escribir en alguna de las 3 secciones restantes (1, 2 y 3) de la memoria flash. Una vez escrito en la memoria flash el fichero (**stratix_top.rbf**) es posible utilizar un conjunto de *dipswitch* los cuales selecciona la sección de memoria flash que será utilizada para la configuración de la FPGA Stratix. La configuración tendrá efecto una vez que se reinicie el computador o se presione el botón de reste (PB3) de la tarjeta de desarrollo.

En la tabla 7.1 se muestra qué posición debe tener cada *dipswitch* para acceder a una sección específica de la memoria flash.

Para el tercer y último caso también se ejecuta la aplicación proporcionada por el *kit* de desarrollo. La diferencia consiste en que la aplicación tiene un botón para realizar la configuración de la Stratix. Esto significa que utilizando la aplicación primero se realiza la escritura en la memoria flash, seleccionando una sección de la memoria, y posteriormente se realiza la escritura. En la figura 7.2 se muestra una pantalla de la aplicación del *kit*, de la tarjeta de desarrollo Stratix. En la figura se puede observar que existen varios botones para distintas funciones. Uno de ellos es para buscar y seleccionar el archivo (**stratix_top.rbf**) para escribir en la memoria flash. Un segundo botón es para seleccionar la sección de la memoria flash donde será escrito el archivo (**stratix_top.rbf**). Un tercer botón es para indicar que se inicie la escritura en la memoria flash y cuarto botón es para solicitar se ejecute la configuración de la Stratix.

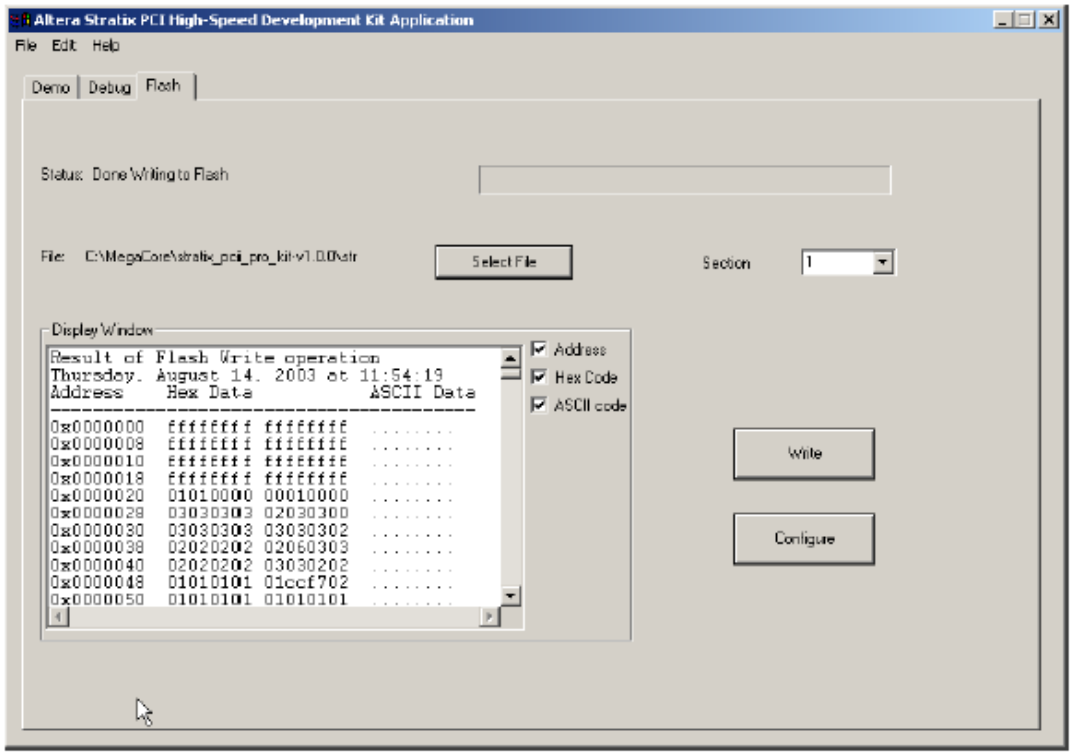


Figura 7.2: *Se muestra la interfaz gráfica para programar y configurar la Stratix, mediante la memoria flash, utilizando la aplicación del* kit *de desarrollo Stratix PCI.*

Al reiniciarse la computadora o la tarjeta de desarrollo, el sistema operativo detectará un nuevo dispositivo hardware y por lo tanto el sistema automáticamente

ejecuta el software para seleccionar o instalar el controlador apropiado para el mismo.

Configuración mediante la interfaz JTAG

Este tipo de configuración se realiza una vez encendida la tarjeta de desarrollo Stratix PCI. El dispositivo Stratix se configura mediante la interfaz JTAG. La interfaz JTAG permite a Quartus II software descargar el archivo (**stratix_top.sof**) dentro del dispositivo Stratix utilizando el cable USB Blaster II y a través de la FPGA EPM3256.

Para llevar a cabo esta configuración es necesario realizar una secuencia de pasos explicados detalladamente en [Cor05c]. Entre los pasos están:

1. Conectar el cable USBBlaster II, entre la computadora y la tarjeta de desarrollo.
2. Ejecutar el software Quartus II y abrir la ventana de programación, dentro de **Tools menu**.
3. Seleccionando el hardware a programar o configurar, en este caso la FPGA EP1S60F1020C6.
4. Seleccionar el archivo a descargar, en este caso el **stratix_top.sof**.
5. Iniciar la programación en **Start**.
6. Cerrar Quartus II software y reiniciar el sistema.

Al reiniciarse la computadora el sistema operativo detectará un nuevo dispositivo hardware y por lo tanto el sistema automáticamente ejecuta el software que seleccionará e instalará el controlador apropiado. El controlador apropiado fue diseñado previamente para esta aplicación, en otro caso, si no se cuenta con el controlador se deberá ejecutar la herramienta para diseñar el controlador.

7.4. Verificación hardware del diseño

Uno de los problemas que suele aflorar cuando se desarrolla un nuevo hardware para ordenadores, consiste en que el software asociado no puede ser programado o puesto a punto hasta que la implementación física del hardware sea, hasta cierto punto, operativa. Hasta este momento ya se cuenta con la tarjeta de desarrollo configurada para la aplicación desarrollada. Sin embargo, para llevar acabo la verificación hardware, esto es el correcto funcionamiento del sistema, es necesario crear el controlador que maneje la interfaz PCI incorporada en el sistema diseñado.

Una vez que se tenga el controlador será posible realizar la verificación del sistema y una serie de pruebas a la aplicación desarrollada. Para la verificación del sistema es necesario desarrollar un software que permita realizar transferencias de datos, vía el bus PCI, hacia la tarjeta de desarrollo Stratix y desde la tarjeta. De esta manera al leer los datos resultantes, de la tarjeta de desarrollo, es posible realizar comparación y por lo tanto verificar el correcto funcionamiento del sistema. A este programa se le nombró `Interfaz_FO`.

Ahora bien, para verificar el correcto funcionamiento y saber que los resultados que proporciona el sistema implementado son correctos es necesario utilizar imágenes de prueba de las cuales es necesario conocer el valor del flujo óptico resultante. Existen imágenes utilizadas comúnmente en la literatura, como por ejemplo, en [BFB94], [BB95] que incluso permiten descargarlas vía Internet [1][2]. Básicamente hay dos tipos de imágenes: las sintéticas y las naturales. Las imágenes sintéticas son utilizadas para determinar técnicamente errores de cálculo, pues se crean imágenes con características establecidas y con conocimiento del movimiento dentro de la secuencia de imágenes. Las otras imágenes, llamadas naturales, pueden ser capturadas en un laboratorio, con ambientes de iluminación y movimientos controlados o bien aquellas que son capturadas fuera de un laboratorio pero que sirven para visualizar el movimiento y analizarlo empíricamente.

Sin embargo, en esta sección sólo se realizará una la verificación del correcto funcionamiento de la arquitectura, esto es, que permita escribir, leer y procesar la información sin importar que los datos resultante sean correctos o incorrectos. De esta manera, para continuar con la verificación hardware de un correcto funcionamiento del sistema, es necesario crear primero el controlador que maneje la interfaz diseñada.

7.4.1. El controlador

Un controlador es un componente que realiza el proceso de comunicación entre un recurso software y un recurso hardware. En este caso realiza la comunicación entre el sistema operativo - bus PCI - tarjeta de desarrollo Stratix PCI. Es común escuchar que el uso del bus PCI es fácil de programar y operar, dada su característica plug&play. Hasta cierto punto es verdad, siempre y cuando se tenga el controlador del hardware que se va a utilizar. Pero si se tiene un dispositivo que puede configurar,

[1] ftp.csd.uwo.ca/pub/vision

[2] http://www.cs.otago.ac.nz/research/vision/

sus registros BAR's, con diferentes valores o diferentes características, es necesario contar con un controlador para cada tipo de configuración establecida.

Solucionar el problema de generar un controlador para cada configuración de dispositivo no es una tarea fácil y peor aún cuando se requiere diseñar controladores para los sistemas operativos de Microsoft, Windows NT/ 2K/ XP, donde su alta "seguridad"no permite operaciones en los puertos de E/S en todos los niveles de la aplicación. En los sistemas operativos anteriores, Windows 95/98, no existía este problema. En este caso, utilizando el sistema Windows XP, es necesario utilizar herramientas más complejas para el desarrollo de controladores con la finalidad de que se tenga acceso a los puertos de E/S.

Un controlador se ejecuta en el nivel más privilegiado del núcleo del procesador. Cada interfaz PCI puede tener diferentes características en los registros de configuración, de tal manera que una vez diseñada la interfaz PCI se requiere contar con el controlador específico para que realice la comunicación entre la interfaz PCI y el sistema operativo. Existen varios caminos para abordar el diseño de un controlador los cuales se diferencian en el coste y tiempo del diseño.

El camino más fácil, pero el más costoso, es utilizar la herramienta de desarrollo Jungo WinDriver[3]. La simplicidad de este software de desarrollo hace posible diseñar un controlador en sólo unos minutos. Con el solo hecho de ejecutar el wizard de WinDriver, detecta automáticamente los dispositivos conectados en el bus PCI. De ese grupo de dispositivos se elige la tarjeta de Altera PCI asignando un nombre al dispositivo y creándose un archivo (**.inf**). Con esto es suficiente para indicar a Windows que esta tarjeta deberá usar los controladores de WinDriver. Se sale del wizard y se instala el controlador. Nuevamente se ejecuta el wizard pero esta vez para el desarrollo de la interfaz con el usuario que permita el acceso a la tarjeta PCI. El costo de este producto es elevado para equipos de trabajo no lucrativos, dado que ronda los $2.000USD. Sin embargo, es posible utilizar un *shareware* por 30 días, tiempo necesario para crear el controlador del trabajo desarrollado.

Otra alternativa consiste en utilizar Microsoft Windows DDK. Esta herramienta es un poco más compleja de utilizar pero mucho más económica, tan solo un 10 % del costo del WinDriver. Su desventaja es que sólo sirve para trabajar bajo Windows. A diferencia Jungo puede trabajar con varias plataformas como: Windows, Linux, Solaris o VxWorks. Sin embargo, también existen otras herramienta para el

[3]http://www.jungo.com

desarrollo de controladores menos conocidas[4].

Una alternativa un tanto radical sería utilizar la tarjeta de desarrollo Stratix, una vez configurada, bajo ambiente Linux. Esta opción disminuiría costes y facilitaría el diseño del controlador. Pues según el libro *Linux Device Drivers, 2nd Edition, Online Book*[5] crear un controlador bajo Linux es más fácil y además es posible acceder al sistema en menos tiempo.

Una vez que ya se cuenta con el controlador de la tarjeta de desarrollo Stratix PCI, se procede a diseñar un programa en C++ Builder para que actúe como Maestro e inicie el procesado de las imágenes.

7.4.2. Interfaz de usuario

Una vez desarrollado el controlador es necesario desarrollar un programa que actúe como Maestro e inicie el procesado de las imágenes. El programa permitiría además obtener información relevante como tiempos de procesado y determinar la magnitud del error.

Sin embargo, el objetivo primordial en esta sección será sólo determinar que la arquitectura diseñada pueda escribir, leer y procesar la información sin importar que los datos resultantes sean correctos o incorrectos. De esta manera la interfaz deberá ser capaz de solicitar el control del bus PCI, cuando lo requiera el sistema. Después, ya que posea el control del bus, realizará la adquisición de las imágenes mediante una cámara digital y las escribirá en la tarjeta de desarrollo Stratix PCI. En la tarjeta de procesará la información y deberá realizar una petición de lectura al bus PCI para escribir los resultados.

Entonces, de manera formal, el programa necesario realiza tres funciones fundamentales: obtener una imagen nueva de la cámara digital, transferir dicha imagen a la tarjeta de desarrollo Stratix para su procesado y después recuperar los datos, ya procesados, de la tarjeta de desarrollo. Una alternativa muy común y obligatoria en ciertos casos es sustituir la primera etapa del programa. En vez de obtener las imágenes directamente de la cámara es posible leerlas de la memoria del computador que previamente fueron capturadas o creadas, en el caso de las imágenes sintéticas. Después de ser leídas, de la memoria del computador, serán escritas en la tarjeta de desarrollo para que sean procesadas. Como se dijo anteriormente, esta alternativa es

[4]http://home.comcast.net/ the_oracle2/driver/software.htm
[5]http://www.xml.com/ldd/chapter/book

viable y necesaria en el caso de querer determinar el error del flujo óptico calculado. Pues para poder realizar pruebas, una evaluación de los resultados obtenidos, es necesario conocer el flujo óptico real, de las secuencias de imágenes. En esos casos se utilizan secuencias de imágenes sintéticas que se pueden descargar de Internet.

De esta manera se considera innecesario realizar un programa que capture las imágenes directamente de la cámara pues basta con procesar la información *off-line* para realizar pruebas concluyentes para este trabajo. Sin embargo, se deja la consideración para el caso de querer realizar la aplicación *on-line*. Aquí se propone realizar la verificación de la arquitectura utilizando dos programas, que evaluarán la arquitectura de forma creciente. Los programas que se proponen realizan la siguiente función:

1. Primero se realiza un programa que permita evaluar sólo una parte de la arquitectura. El programa consiste en escribir en la tarjeta de desarrollo Stratix PCI un par de imágenes previamente capturadas, localizadas en la memoria del computador, y posteriormente son leídas, de la tarjeta, las mismas imágenes. La finalidad es evaluar sólo el funcionamiento del bus PCI y del controlador de la DDR SDRAM.
2. El segundo programa consiste en verificar el funcionamiento del resto del sistema, en particular de la arquitectura realizada. De esta manera el programa escribe las imágenes, localizadas en la memoria del computador, en la memoria DDR SDRAM. Una vez que las imágenes se encuentran en la tarjeta de desarrollo son procesadas y los datos resultantes, que se encuentran almacenados en la memoria DDR SDRAM, deben ser leídos por el programa para escribirlos en el computador. Con este programa es posible determinar el rendimiento, precisión y la viabilidad del sistema de procesado guiado por cambios aquí desarrollado.
3. Un último programa consistiría en realizar el procesado del flujo óptico guiado por cambios *on line*. De esta manera las imágenes capturadas por la cámara digital son enviadas inmediatamente a la memoria DDR SDRAM, localizada en la tarjeta de desarrollo. El resto del programa realiza una función similar al segundo programa diseñado. Este programa es posible omitir, debido a que el segundo programa realizado permite verificar y obtener información concluyente del funcionamiento de la arquitectura diseñada. De hecho existen pruebas determinantes que sólo con el segundo programa es posible evaluar y que en muchas ocasiones, incluso con secuencias de imágenes naturales, es necesario utilizar secuencias de imágenes previamente evaluadas en otros trabajos.

Programa No.1

Este primer programa consiste en verificar que exista una correcta comunicación entre el bus PCI y la memoria DDR SDRAM. De esta manera también se comprueba el funcionamiento del controlador diseñado. El programa diseñado efectúa la solicitud para tomar el control del bus PCI y realizar las transacciones necesarias. En la interfaz se crea una pantalla en la que se leerán dos imágenes, de una secuencia. Las imágenes serán leídas de la memoria del computador y escritas en la memoria DDR SDRAM. Posteriormente serán leídas y desplegadas en pantalla. En la figura 7.3 se muestra la pantalla en la cual se leen del disco duro dos imágenes, de una secuencia típica utilizada, que son escritas en la memoria DDR SDRAM. Después, en la misma figura, en la parte inferior se muestran las mismas imágenes leídas de la memoria DDR SDRAM.

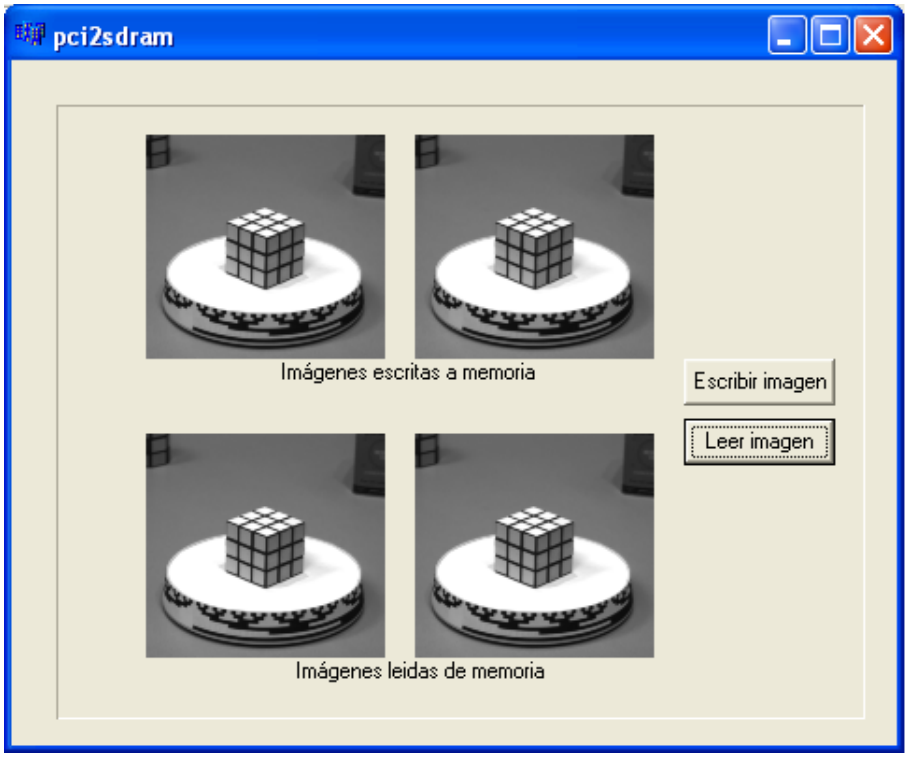


Figura 7.3: *Interfaz del usuario en la que se despliegan las dos imágenes que son escritas en la memoria DDR SDRAM. Después son leídas y desplegadas de la misma manera.*

Como se dijo anteriormente, este programa simplemente cumple con el propósito de verificar el correcto funcionamiento de la interfaz PCI y de su controlador. En

particular para efectuar transacciones de escritura/lectura entre el computador y la memoria DDR SDRAM. Las imágenes utilizadas en este programa son la 8 y 9 de la secuencia *Rubic*, que tienen un tamaño de 240×256 píxeles con 256 tonos de gris. La secuencia de imágenes se obtuvo de la Universidad de Western Ontario, Departamento de Ciencias de la Computación[6].

Programa No.2

En este segundo programa, que tiene un alcance mayor, también se realiza la lectura de dos imágenes consecutivas de una secuencia de imágenes localizada en el disco duro. En este programa ya se permite seleccionar las imágenes a procesar. Las imágenes son escritas en la memoria DDR SDRAM, de la tarjeta de desarrollo Stratix. En la figura 7.4 se pueden observar las dos imágenes a ser procesadas, que son la 8 y 9 de la secuencia *Rubic*. También se visualiza en la misma figura la imagen resultante, leída de la memoria DDR SDRAM, y además se indica en donde se almacena dicha imagen y un fichero que contendrá el resto de la información que se haya calculado, como es el caso del error y tiempo de procesado.

Es importante resaltar que la imagen resultante se obtiene de dos imágenes, que contienen las componentes de la velocidad (U, V) y de la tabla de cambios LUT principal. La LUT sirve para leer correctamente las componentes de la velocidad puesto que sólo se leyeron de la memoria DDR SDRAM las componentes de la velocidad que pertenecen a los píxeles que si se procesaron o bien que sí cambiaron. De esta manera se reconstruye la imagen de los vectores de flujo óptico.

Con esto es posible concluir que finalmente se realiza el cálculo del flujo óptico guiado por cambios mediante una arquitectura de flujo de datos. El proceso se realizó con las imágenes consecutivas 8 y 9 de la secuencia *Rubic*, que tienen un tamaño de 240×256 píxeles de 8 bits. El procesado guiado por cambios se realizó con un umbral $t_h \geq 2$ y con 10 iteraciones.

El siguiente paso sería evaluar el rendimiento y la precisión del sistema. Esto significa que se tiene que evaluar el sistema utilizando imágenes de las cuales se conozca a priori el flujo óptico real. Este proceso se realiza en el capítulo 8.

[6] ftp.csd.uwo.ca/pub/vision

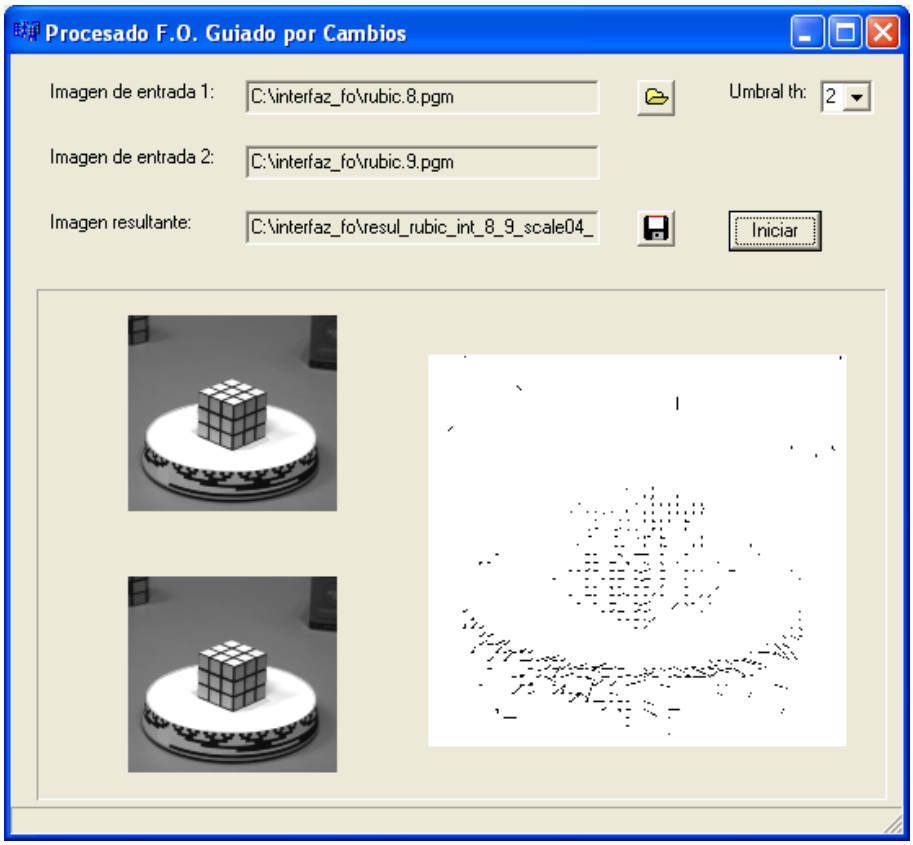


Figura 7.4: *Interfaz del usuario en la que se despliegan las dos imágenes que son escritas en la memoria DDR SDRAM y la imagen resultante de los vectores del flujo óptico.*

Capítulo 8

Resultados experimentales

8.1. Introducción

Para llevar a cabo la validación de la eficiencia del sistema diseñado, es necesario manejar una serie de secuencias de imágenes de las cuales se conozca a priori su flujo óptico. Existen bancos de imágenes, comúnmente utilizados en la literatura, los cuales fueron capturados tanto en ambientes no controlados, como en ambientes un tanto controlados (ambientes cerrados). Además de estos tipos de secuencias de imágenes existen otras secuencias llamadas sintéticas. Las imágenes sintéticas son utilizadas para evaluar con una mayor precisión el error del sistema respecto de los cambios reales existentes en la secuencia de imágenes. Todos los bancos de imágenes son utilizados comúnmente para el análisis del movimiento y más particularmente para evaluar el movimiento aparente o flujo óptico obtenido. De esta manera, con el fin de evaluar el sistema propuesto sólo es necesario utilizar algunas secuencias de imágenes. De hecho serán utilizadas las secuencias más comunes utilizadas en la bibliografía.

Las secuencias de imágenes utilizadas para este propósito fueron obtenidas de la Universidad de Western Ontario, Departamento de Ciencias de la Computación[1] y del grupo de investigación de Visión por Computador, de la Universidad de Otago[2].

Por otro lado, debido a que se va a manejar una gran cantidad de datos es necesario realizar un programa en C++ con el fin de facilitar la tarea de evaluación del sistema hardware diseñado. El programa, utilizado en el capítulo 4, permite

[1] ftp.csd.uwo.ca/pub/vision

[2] http://www.cs.otago.ac.nz/research/vision/

realizar diferentes evaluaciones del diseño, entre las que están:

1. **Evaluación visual.** Se realiza una comparación visual entre las imágenes de flujo óptico obtenidas y las imágenes de flujo óptico conocidas. Para este propósito, comúnmente son utilizadas las secuencias de imágenes naturales puesto que en ellas difícilmente se conoce el valor exacto del flujo óptico.
2. **Evaluación formal.** Se realiza un análisis del error cuantitativo del flujo óptico. En este caso se utilizan imágenes sintéticas, pues en ellas se conoce perfectamente el flujo óptico existente.

El proceso de evaluación se inicia realizando una comparación visual de las imágenes obtenidas y del tiempo necesario para el procesado. Para este propósito se consideran dos secuencias de imágenes reales o naturales y una secuencia sintética. De las dos secuencias de imágenes naturales una de ellas fue capturada en un ambiente cerrado, con cierto control de la iluminación, como es el caso de la secuencia *Rubic*. La otra secuencia fue capturada por una cámara que registra la escena exterior de un edificio, esta es la secuencia *taxi*. La escena sintética, llamada *sphere*, es utilizada sólo con propósito comparativo. También se realiza un estudio del número de píxeles que permanecen constantes en el tiempo, entre pares de imágenes consecutivas, dentro de cada una de las escenas estudiadas.

Posteriormente a la evaluación visual, se realiza la evaluación formal. Para la evaluación formal es necesario manejar secuencias de imágenes de las que se conozca el flujo óptico real. La intención de conocer el flujo óptico real es poder determinar el error entre el flujo óptico real y el flujo óptico calculado por el sistema diseñado, el cual realiza el procesado de imágenes guiado por cambios. Adicionalmente se realiza una comparación entre el flujo óptico calculado mediante el algoritmo original, utilizando un computador que procese la información en coma flotante, y el flujo óptico calculado por la arquitectura aquí propuesta, para así determinar el error relativo del sistema. Para esta evaluación se utilizan las secuencias *Diverging tree* (*treed*), *Traslating tree* (*treet*) y la secuencia *square2*.

Con ambas evaluaciones se intenta obtener resultados concluyentes del rendimiento y la precisión de la arquitectura propuesta.

8.2. Evaluación visual y aumento del rendimiento

Para llevar a cabo esta evaluación se utilizaron dos secuencias de imágenes naturales y una tercera secuencia de imágenes del tipo sintética. Las secuencias de imágenes fueron: *taxi*, *Rubic* y *sphere*. En todos los casos, las secuencias de imágenes son en blanco y negro con 256 tonos de gris (píxeles de 8 bits).

Las secuencias de imágenes utilizadas se eligieron debido a que básicamente son las que representan los tres tipos de imágenes que puede analizar y procesar un sistema como el aquí propuesto. La secuencia *taxi* es de tipo natural y fue capturada en el exterior de un edificio, de tal manera que esta secuencia contiene todos los factores o ruidos que pueden existir en la vida real. La secuencia *Rubic* también es del tipo natural pero con la diferencia de que fue capturada en un lugar cerrado y por lo tanto existe un ambiente más controlado, en cuanto a la iluminación y movimientos dentro de la escena. Finalmente la secuencia *sphere* es del tipo sintética y en ella sólo se reflejan los movimientos existentes en la escena y se puede decir que es inmune a los ruidos naturales existentes cuando las imágenes son capturadas.

En la figura 8.1 se muestra una gráfica que ilustra el comportamiento de los cambios existentes entre 11 pares de imágenes consecutivas, correspondientes a las tres secuencias de imágenes utilizadas en esta sección. Es posible ver que la secuencia *sphere*, que es una secuencia de imágenes sintéticas, muestra un porcentaje casi constante de píxeles que no cambiaron (píxeles de pares de imágenes consecutivas). Todos esos píxeles no son procesados en el sistema. En sentido opuesto la secuencia *taxi*, que son imágenes naturales, presenta un porcentaje de cambios aleatorio. Además de eso, tiene un menor porcentaje de píxeles constantes entre pares de imágenes consecutivas. Por lo tanto el sistema procesa una mayor cantidad de píxeles. Finalmente, la secuencia natural *Rubic* representa un punto intermedio entre las secuencias *sphere* y *taxi*. La razón es que esta secuencia natural fue capturada en un ambiente, hasta cierto punto, controlado.

De cada secuencia de imágenes se tomó un par de imágenes para calcular el flujo óptico de acuerdo a las opciones que se enumeran a continuación:

1. La arquitectura hardware diseñada, la cual procesa la información guiada por cambios denominada FOGCH.
2. Un computador que realiza el cálculo del flujo óptico utilizando aritmética de coma flotante y la estrategia del procesado guiado por cambios llamada FOGCS.
3. Un computador que realiza el cálculo del flujo óptico utilizando aritmética de

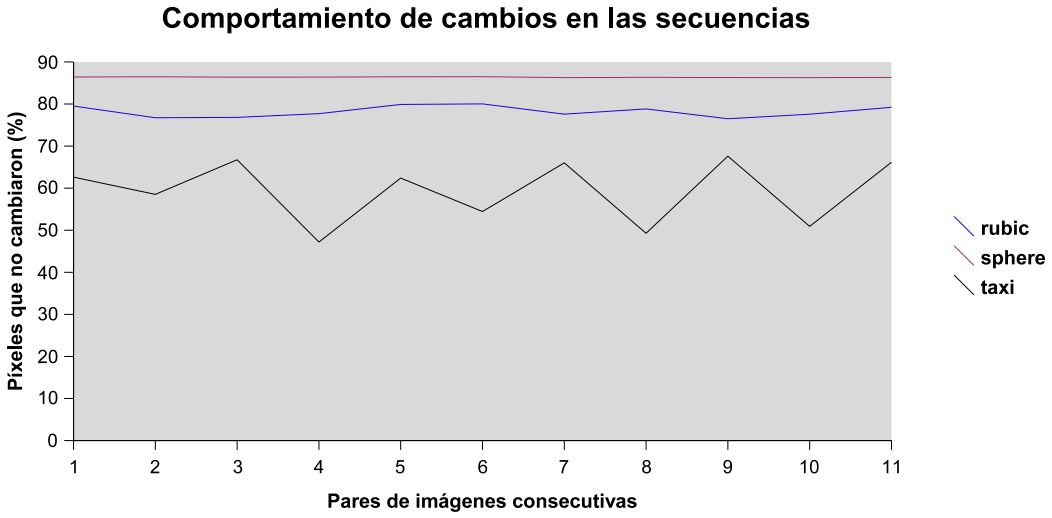


Figura 8.1: *Porcentaje de cambios existentes entre 11 pares de imágenes consecutivas, de las secuencias de imágenes* Rubic, sphere *y* taxi.

coma flotante con el algoritmo original de Horn, el cual procesa todos los píxeles del par de imágenes (FOFP).

En la mayoría de los casos se utilizó un umbral de $t_h \geq 1$ con 10 iteraciones. El computador utilizado para realizar los cálculos en coma flotante posee un Procesador Intel Xeon a 3 GHz, 512 MByte en RAM y Disco duro de 80 GByte. El computador trabaja sobre el Sistema Operativo de Microsoft (Windows 2000).

Secuencia *taxi*

Esta secuencia de imágenes, de tamaño 190×256 píxeles, es de tipo natural y ha sido capturada en el exterior. Por tal razón aumenta la posibilidad de que sean más los píxeles que cambien, entre pares de imágenes consecutivas. Ese hecho conlleva a que sean más los píxeles a ser procesados. Para ilustrar los objetos que intervienen en la escena de la secuencia utilizada se presenta la figura 8.2. En ella se muestra un par de imágenes, en particular las imágenes 3 y 4, de la secuencia original *taxi*.

Para llevar a cabo la evaluación se estableció el umbral para la detección de cambios a $t_h \geq 1$ y se realizaron 10 iteraciones ($k = 10$). Las imágenes de flujo óptico obtenidas se muestran en la figura 8.3. La figura a) representa el flujo óptico procesando todos los píxeles (FOFP) y utilizando el algoritmo de Horn original. La figura b) muestra el flujo óptico calculado por software utilizando la técnica de

Figura 8.2: *Par de imágenes consecutivas originales, de la secuencia* taxi*, utilizadas en el calcular el flujo óptico.*

procesado guiado por cambios (FOGCS). Finalmente la figura c) muestra el flujo óptico calculado por la arquitectura hardware diseñada que utiliza la estrategia de procesado guiado por cambios (FOGCH). Es posible observar que la diferencia visual entre las tres imágenes obtenidas es mínima, e incluso se podría decir que son casi idénticas, todo eso a pesar de que las dos primeras imágenes son obtenidas utilizando aritmética de coma flotante, a diferencia de la tercera imagen donde se utiliza una aritmética entera. De hecho la diferencia más notable, que por cierto no es visible pero si medible, es el tiempo necesario para efectuar el cálculo del flujo óptico, los cuales se despliegan en el cuadro 8.1.

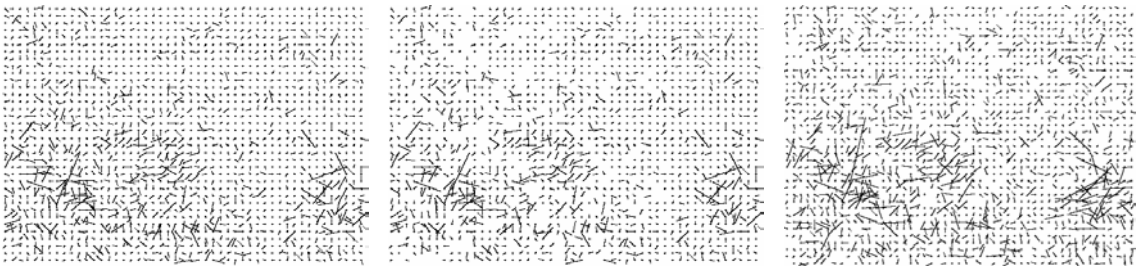


Figura 8.3: *Flujo óptico obtenido con un umbral* $t_h \geq 1$ *y con 10 iteraciones: a) flujo óptico procesando todos los píxeles (FOFP), b) flujo óptico guiado por cambios software (FOGCS) y c) flujo óptico guiado por cambios hardware (FOGCH).*

Analizando las imágenes de la figura 8.3 es posible observar que las imágenes de los vectores de flujo óptico muestran información un tanto difusa. De hecho, en las tres imágenes obtenidas existen muchos píxeles que presentan movimiento aparente. Esto genera confusión cuando es necesario identificar qué objetos realmente se están

Tiempo de procesado (*ms*)		
FOFP	**FOGCS**	**FOGCH**
90	80	21,46

Cuadro 8.1: *Tiempos de procesado para el cálculo del flujo óptico FOFP, FOGCS y FOGCH con 10 iteraciones y* $t_h \geq 1$*, para el par de imágenes consecutivas "taxi".*

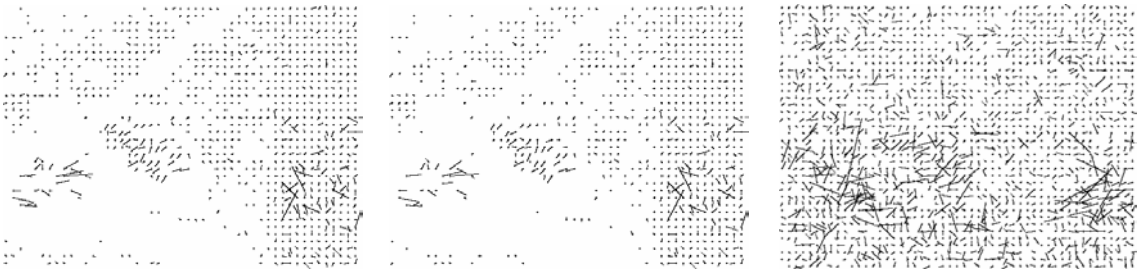
movimiento dentro de la escena. En la escena sólo existen 4 objetos en movimiento. Uno de los objetos, es un auto blanco en el centro de la imagen que ingresa a una calle a su derecha, un segundo objeto es un auto que va detrás del auto blanco. Un tercer objeto es un auto oscuro que viene en sentido contrario a los dos autos mencionados anteriormente y el cuarto objeto, en movimiento, es una persona que se encuentra en la esquina de la calle, frente al auto blanco, que apenas es identificable. El resto de los píxeles que presentan cambios en la intensidad en realidad es ruido o cambios de iluminación debidos a factores externos, que pueden ser producidos por la cámara o el medio ambiente.

Para evitar este tipo de problema y reducir el número de movimientos, Barron et al. propusieron realizar una modificación al algoritmo original de Horn y Schunck [BFB94]. La modificación consiste en discriminar los datos y sólo desplegar aquellos cuya intensidad de los gradientes (I_x e I_y) supere un valor establecido. Es importante resaltar que ese proceso es una etapa adicional en la que deben ser leídos todos los píxeles del par de imágenes. Esto significa que después de haber calculado el flujo óptico se procede a seleccionar los vectores de flujo óptico que serán visualizados en función de los gradientes (I_x e I_y). La fracción del código, que realiza la discriminación de los vectores de flujo óptico a visualizar, se muestra a continuación:

```
.
.
TAU=5;
//Son calculados los vectores del flujo óptico (U,V).
.
.
for i=2:m
   for j=2:n
     grad2= Ix(i,j)* Ix(i,j)+ Iy(i,j)* Iy(i,j);
       if grad2<=(TAU*TAU)
        U(i,j)=NO_VALUE;
        V(i,j)=NO_VALUE;
       end
   end
end
```

La variable que determina el umbral a considerar, para seleccionar los vectores a ser graficados, es denominada TAU en el código y en la literatura se maneja

como $||\nabla I||$ [BFB94]. Al efectuar la modificación en el código original, del algoritmo de Horn, es posible identificar más fácilmente los objetos que están en movimiento dentro de la escena. De esta manera partiendo de la información obtenida, mostrada en las imágenes de la figura 8.3 y con $||\nabla I|| \geq 5$, se obtienen las nuevas imágenes de vectores de flujo óptico que son mostradas en la figura 8.4.



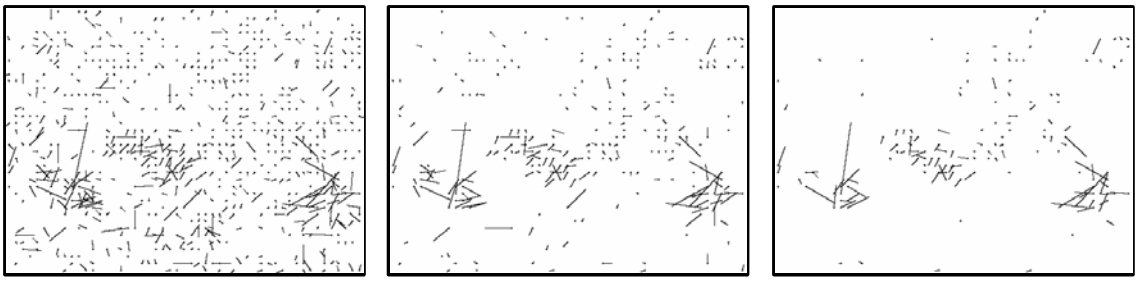
a) **b)** **c)**

Figura 8.4: *Se muestra el flujo óptico obtenido con un $t_h \geq 1$, 10 iteraciones y un $||\nabla I|| \geq 5$. Las figuras muestran: a) flujo óptico procesando todos los píxeles (FOFP), b) flujo óptico guiado por cambios software (FOGCS) y c) flujo óptico guiado por cambios hardware (FOGCH).*

Observando las imágenes de la figura 8.4 es posible apreciar que sólo se obtuvo una mejora en las imágenes a) y b). La razón es porque durante el cálculo se utilizó memoria suficiente para almacenar las imágenes parciales necesarias, como es el caso de las imágenes de los gradientes (I_x, I_y e I_t). En cambio para el caso de la imagen de la figura 8.4 c), que fue obtenida mediante la arquitectura diseñada, las componentes I_x, I_y e I_t son eliminadas una vez que ya no son utilizadas pues son innecesarias una vez calculados los vectores del flujo óptico. De esta manera se deja claro que no es posible efectuar dicho proceso pues se carece de los elementos necesarios para realizar dicho propósito. En una arquitectura diseñada anteriormente [SGFBP06b], sí que es posible realizar dicho post-procesado. Esto es debido a que se almacenan las componentes de los gradientes en memoria y es posible utilizarlas para el post-procesado.

Sin embargo, para la arquitectura diseñada en este trabajo el post-procesado realizado se hace innecesario, pues se cuenta con la posibilidad de modificar el umbral t_h para realizar un propósito análogo. La modificación del umbral t_h tiene un efecto directo en la discriminación de los movimientos aparentes, dando al sistema un grado de inmunidad al ruido. Es importante destacar que el umbral es un factor muy sensible e importante pues con valores grandes es posible que se elimine in-

formación relevante de la escena y con valores muy pequeños se reducen los efectos de inmunidad al ruido. Por tal razón es necesario considerar un umbral pequeño, pero que sirva para dicho propósito. Por lo tanto se realizó una evaluación, de la arquitectura diseñada, con varios umbrales. Los umbrales utilizados fueron $t_h \geq 3$, $t_h \geq 5$ y $t_h \geq 7$, lo que significa que sólo son procesados los píxeles que cambian, en su tono de gris, con una diferencia absoluta igual o mayor a 3, 5 y 7 niveles de gris respectivamente, entre los píxeles del par de imágenes consecutivas. En la figura 8.5 se muestran las imágenes del flujo óptico resultantes, para los tres umbrales utilizados.



a) **b)** **c)**

Figura 8.5: *Flujo óptico obtenido con a)$t_h \geq 3$, b)$t_h \geq 5$ y c)$t_h \geq 7$, en todos los casos se realizan 10 iteraciones y se calcula el flujo óptico utilizando la arquitectura hardware diseñada (FOGCH).*

Además de obtener una mejor visualización de los objetos que realmente están en movimiento dentro de la escena, se logra reducir el tiempo de procesado, obteniéndose un mejor rendimiento del sistema. Esto se debe a que se procesa una menor cantidad de píxeles entre mayor sea el umbral. Por ejemplo, la imagen del flujo óptico de la figura 8.3 c) se obtiene al procesar 40.089 píxeles de cada imagen. Los píxeles procesados son aquellos que presentan un cambio significativo, por encima a un umbral de $t_h \geq 1$, que corresponde a procesar el 82,42 % del total de los píxeles. Al incrementar el umbral a $t_h \geq 3$ se procesan 15.503 píxeles de cada imagen, que representa el 31,87 % del total de los píxeles y se obtiene la imagen mostrada en la figura 8.5 a). Si el umbral se incrementa a $t_h \geq 5$, se procesan 5.960 píxeles que equivale al 12,25 % y se obtiene la figura 8.5 b). Y si el umbral se incrementa hasta $t_h \geq 7$, sólo son procesados 3.316 píxeles que corresponde al 6,817 % del total de los píxeles, obteniéndose la figura 8.5 c).

De esta manera, es posible concluir que entre mayor sea el umbral menor será el número de píxeles a procesar por lo tanto el procesado de la información se realiza en

Umbral $t_h \geq$	**Tiempo de procesado**	**% de píxeles procesados**	**No. de imágenes procesadas**
1	21,46 *ms*	82,42	46,5
3	13,87 *ms*	31,87	72,0
5	10,90 *ms*	12,25	91,7
7	10,11 *ms*	6,81	98,9

Cuadro 8.2: *Tiempo de procesado y porcentaje de píxeles procesados para el cálculo del flujo óptico FOGCH con 10 iteraciones y distintos umbrales (t_h). Se utilizó el par de imágenes "taxi", de tamaño* 190×256 *píxeles.*

un menor tiempo. Esto queda reflejado en el cuadro 8.2. Del cuadro se puede ver que con un $t_h \geq 1$ es posible procesar 46,5 imágenes por segundo (fps). Sin embargo, en función a los resultados visuales obtenidos el umbral que permite reducir la mayor cantidad de ruido, eliminando la menos cantidad de información relevante, es cuando se utiliza un umbral $t_h \geq 5$ y permite procesar hasta 91,7 fps. Es importante aclarar que esta frecuencia de procesado es particularmente para el par de imágenes utilizadas durante el análisis, pudiendo variar ligeramente en otro par de imagen.

Secuencia *Rubic*

La secuencia de imágenes Rubic también es del tipo natural, pero a diferencia de la secuencia *taxi*, fue capturada en un ambiente cerrado. Por lo tanto, hasta cierto punto, el ambiente es controlado. En la figura 8.6 se muestran las imágenes 4 y 5 de la secuencia original *Rubic*, las cuales fueron capturadas a un de tamaño 240×256 píxeles. De la misma manera que en el análisis de las imágenes anteriores, se realizó el calculó el flujo óptico mediante la arquitectura diseñada (FOGCH), también utilizando un computador con la estrategia de procesado del flujo óptico guiado por cambios (FOGCS) y finalmente se hizo el cálculo mediante un computador, pero utilizando el algoritmo original de Horn, en el cual se procesan todos los píxeles de las imágenes (FOFP).

Las imágenes fueron procesadas utilizando un umbral de $t_h \geq 1$ y con $k = 10$ iteraciones. Las imágenes resultantes de los vectores de flujo óptico obtenidos se muestran en la figura 8.7. La imagen de la figura 8.7 a) se obtiene utilizando un computador con aritmética de coma flotante y se procesan todos los píxeles de las dos imágenes consecutivas (FOFP). Esta imagen presenta la mayor cantidad de

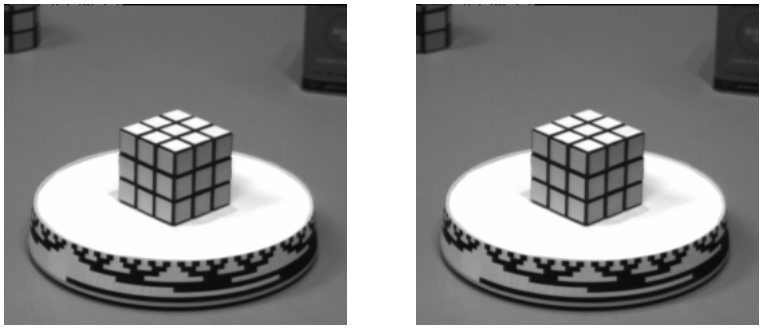


Figura 8.6: *Par de imágenes consecutivas originales de la secuencia* Rubic *utilizadas para calcular el flujo óptico.*

vectores de flujo óptico, sin embargo, muchos de ellos no representan movimientos reales, si no que son movimientos aparentes debidos a cambios en la iluminación. La imagen b) de la misma figura, también fue obtenida mediante un computador con aritmética de coma flotante empleando la técnica de procesado guiado por cambios. La imagen muestra una mejor respuesta lo que indica que posee una mayor inmunidad al ruido. Sin embargo, se puede apreciar que también existen vectores con una magnitud pequeña y que en muchos casos son vectores que representan un movimiento aparente. Finalmente la imagen c) es obtenida utilizando la arquitectura diseñada de aritmética entera, empleando la técnica de procesado guiado por cambios, por lo que presenta una menor cantidad de vectores de flujo óptico respecto a la imagen a) y sólo un poco menos respecto de la imagen b) mostradas en la misma figura.

Figura 8.7: *Flujo óptico obtenido en 10 iteraciones y con un* $t_h \geq 1$*: a)Flujo óptico procesando todos los píxeles (FOFP), b) Flujo óptico guiado por cambios por software (FOGCS) y c)Flujo óptico guiado por cambios por hardware (FOGCH).*

Tiempo de procesado (*ms*)		
FOFP	**FOGCS**	**FOGCH**
141	94	19,71

Cuadro 8.3: *Tiempos de procesado para el cálculo del flujo óptico FOFP, FOGCS y FOGCH con 10 iteraciones, para el par de imágenes "Rubic".*

En la escena real de la secuencia *Rubic* sólo se está moviendo la plataforma con el **cubo** que se encuentra sobre ella. De hecho la plataforma está rotando hacia la izquierda a una velocidad constante y debe quedar claro que ningún otro objeto, dentro de la escena, se está moviendo. Las imágenes b) y c) muestran los vectores de flujo óptico más acordes al movimiento real existente en la escena, en comparación con la imagen a) de la figura 8.7. Por otro lado se observa que se obtiene un mejor rendimiento, al utilizar la arquitectura diseñada, puesto que existe una reducción de los tiempos de procesado como se puede ver en el cuadro 8.3. La razón es que se procesa una menor cantidad de píxeles y por lo tanto se obtiene el resultado en menor tiempo. Por ejemplo, en el caso de la imagen de flujo óptico obtenida, de la figura 8.7 c), se procesaron 26.730 píxeles de cada imagen. Los píxeles procesados son sólo los que presentaron un cambio significativo, por encima a un umbral $t_h \geq 1$, que corresponde a procesar el 43,50 % de un total de 14.880 píxeles.

En el caso de que se realice, como en el caso anterior, la modificación al algoritmo original de Horn, haciendo $||\nabla I|| \geq 5$, con la finalidad de identificar más fácilmente los objetos que están en movimiento, dentro de la escena y partiendo de la información obtenida, mostrada en las imágenes de la figura 8.7, es posible obtener unas nuevas imágenes de vectores de flujo óptico mostradas en la figura 8.8.

De las imágenes de la figura 8.8 se puede ver que sólo se obtiene una mejora en las imágenes a) y b). La razón es que durante el procesado se almacenaron las imágenes de los gradientes (I_x, I_y e I_t). Por el contrario, para el caso de la imagen c), que es obtenida mediante la arquitectura diseñada, las componentes I_x, I_y e I_t son eliminadas al obtenerse los vectores del flujo óptico finales. Por tal razón, al carecer de los elementos necesarios para realizar dicho propósito, no es posible efectuar el mismo proceso realizado a la información obtenida mediante el FOFP y FOGCS. Sin embargo, en la arquitectura desarrollada en [SGFBP06b], donde si se almacenan las componentes de los gradientes en la SDRAM, es posible realizar dicho proceso y se obtiene una imagen casi idéntica a las anteriores, como se puede observar en la figura 8.9.

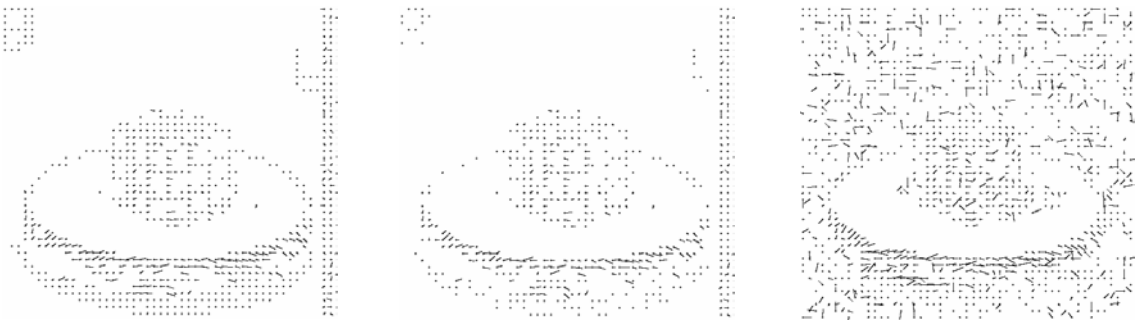


Figura 8.8: *Flujo óptico obtenido con un* $t_h \geq 1$*, 10 iteraciones y un* $||\nabla I|| \geq 5$*: a) flujo óptico procesando todos los píxeles (FOFP), b) flujo óptico guiado por cambios software (FOGCS) y c) flujo óptico guiado por cambios hardware (FOGCH).*

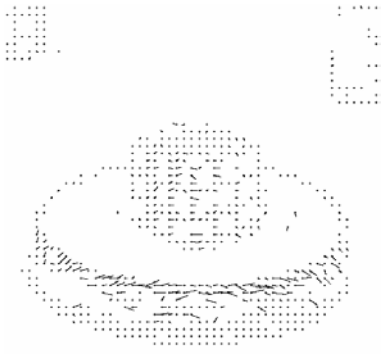
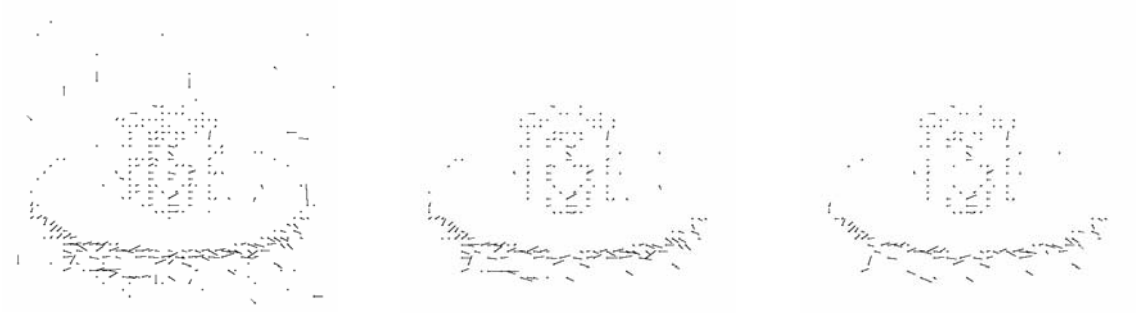


Figura 8.9: *Flujo óptico obtenido mediante la arquitectura diseñada en (FPL06), con un* $t_h \geq 1$*, un* $||\nabla I|| \geq 5$ *y 10 iteraciones.*

En caso de querer realizar este tipo de proceso en la arquitectura diseñada en este trabajo, es necesario realizar algunas modificaciones a la arquitectura. Las modificaciones implicarían un incremento de recursos, en particular de memoria, y además un incremento en el tiempo de procesado. Esto último surge por la necesidad de tener que escribir las componentes de los gradientes en la memoria SDRAM, y después escribir dichas componentes en la memoria del computador. Todo esto es para que puedan ser graficadas utilizando la restricción de un $||\nabla I|| \geq 5$.

La razón del no requerir estas componentes o bien realizar este tipo de proceso es porque en la arquitectura diseñada es posible modificar el umbral t_h, para la detección de cambios, que realiza un propósito análogo. La diferencia, en modificar el umbral t_h, es que se tiene un efecto directo en la discriminación de los movimientos aparentes, dando al sistema un grado de inmunidad al ruido, y ayudando a reducir

el tiempo para la obtención de los vectores de flujo óptico. Como se indicó anteriormente el umbral es un factor muy sensible e importante, pues con valores grandes es posible que se elimine información relevante de la escena y con valores muy pequeños se reducen los efectos de inmunidad al ruido. En este caso se realizó una evaluación, de la arquitectura diseñada, con los umbrales de: $t_h \geq 2$, $t_h \geq 3$ y $t_h \geq 4$. Los umbrales indican que serán procesados aquellos píxeles que superen, en niveles de tono de gris, una diferencia absoluta igual o mayor a 2, 3 y 4 niveles de gris, entre los píxeles del par de imágenes consecutivas. En la figura 8.10 se muestran las imágenes del flujo óptico resultantes, para los tres umbrales utilizados.



a) **b)** **c)**

Figura 8.10: *Flujo óptico obtenido con a)$t_h \geq 2$, b)$t_h \geq 3$ y c)$t_h \geq 4$, en todos los casos se realizan 10 iteraciones y se calcula el flujo óptico utilizando la arquitectura hardware diseñada (FOGCH).*

De las imágenes mostradas en la figura 8.10, se puede observar que es posible obtener una mejor respuesta modificando ligeramente el umbral. Pero además entre mayor es el umbral menor es el tiempo necesario para obtener los vectores del flujo óptico. Esto queda claramente reflejado en el cuadro 8.4. En función a los resultados visuales obtenidos el umbral que permite reducir la mayor cantidad de ruido, eliminando la menos cantidad de información relevante, es cuando se utiliza un umbral $t_h \geq 2$ y con este umbral es posible procesar hasta 73,5 fps. Igualmente que en la secuencia *taxi*, en este caso, la frecuencia de procesado es particular para cada par de imágenes utilizadas durante el análisis y puede variar ligeramente entre un par de imágenes y otro.

Sin embargo, no por lograr un mejor tiempo de respuesta del sistema se debe establecer un umbral grande, puesto que por ejemplo, en este caso con un umbral $t_h \geq 2$ o incluso con un $t_h \geq 3$ es más que suficiente. Pero para un $t_h \geq 4$ se pierde información que puede ser valiosa, en ciertos casos. Los tiempos de procesado se

Umbral $t_h \geq$	**Tiempo de procesado**	**% de píxeles procesados**	**No. de imágenes procesadas**
1	19,71 *ms*	43,50	50,7
2	13,59 *ms*	11,53	73,5
3	12,92 *ms*	7,87	77,3
4	12,67 *ms*	6,56	78,9

Cuadro 8.4: *Tiempo de procesado y porcentaje de píxeles procesados para el cálculo del flujo óptico FOGCH con 10 iteraciones y distintos umbrales (t_h). Se utilizó el par de imágenes "Rubic", de tamaño* 240×256 *píxeles.*

reducen debido a que al utilizar el umbral $t_h \geq 2$ se procesa sólo el 11,53 %, para un umbral de $t_h \geq 3$ se procesa un 7,87 % y al utilizar un umbral $t_h \geq 4$ se procesa tan sólo el 6,56 %, del total de los píxeles que son más de 61 mil por cada imagen.

Secuencia *sphere*

Finalmente, la secuencia *sphere* es evaluada siguiendo el mismo procedimiento que las dos secuencias anteriores. La diferencia entre las dos secuencias anteriores y esta secuencia es que es de tipo sintética. En la figura 8.11 se muestran las imágenes 1 y 2, de la secuencia *sphere* a blanco y negro y de tamaño 200×200 píxeles. La secuencia es una esfera que presenta un movimiento de rotación de derecha a izquierda, no es de un color uniforme y se encuentra sobre un fondo de líneas. Los vectores del flujo óptico real de esta secuencia, son los mostrados en la imagen de la figura 8.12.

Las imágenes resultantes mostradas en la figura 8.13, son obtenidas al calcular el flujo óptico mediante la arquitectura diseñada, la cual procesa la información guiada por los cambios (FOGCH), utilizando un computador y el procesado guiado por cambios (FOGCS) y utilizando un computador pero procesando todos los píxeles de las imágenes (FOFP). En todos los casos se utilizó un umbral de $t_h \geq 1$ y con $k = 10$ iteraciones.

Además de obtener una respuesta visual aceptable, al procesar el par de imágenes utilizando la arquitectura diseñada, también se logró un mejor rendimiento. Esto se puede observar en el cuadro 8.5 donde se muestran los tiempos de procesado. La reducción de los tiempos se debe a que se procesa una menor cantidad de píxeles y

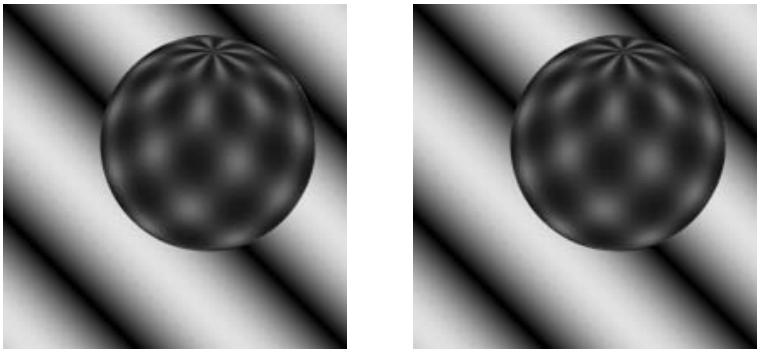


Figura 8.11: *Par de imágenes consecutivas originales de la secuencia* sphere *utilizadas para calcular el flujo óptico.*

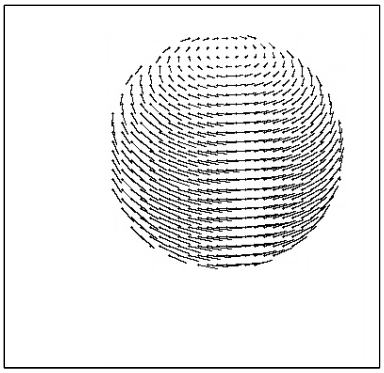


Figura 8.12: *Imagen real de vectores de flujo óptico para un par de imágenes consecutivas de la secuencia* sphere.

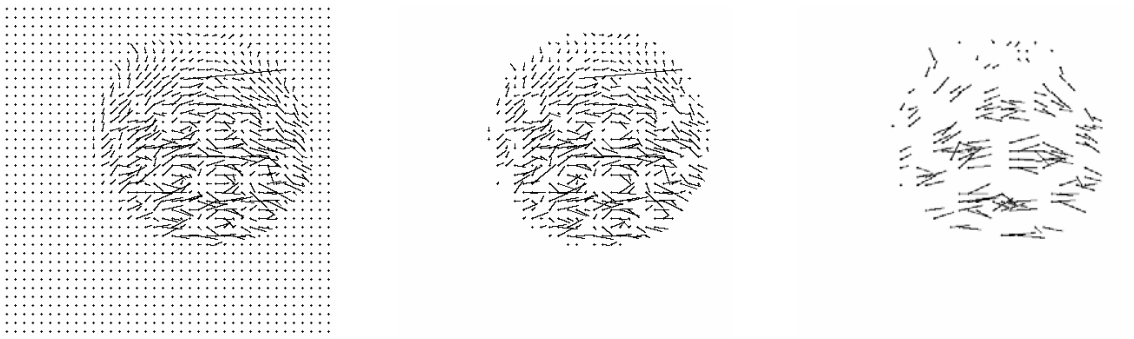


Figura 8.13: *Flujo óptico obtenido en 10 iteraciones y con un* $t_h \geq 1$*: a)Flujo óptico procesando todos los píxeles (FOFP), b)Flujo óptico guiado por cambios por software (FOGCS) y c)Flujo óptico guiado por cambios por hardware (FOGCH).*

Tiempo de procesado (*ms*)		
FOFP	**FOGCS**	**FOGCH**
94	31	12,9

Cuadro 8.5: *Tiempos de procesado para el cálculo del flujo óptico FOFP, FOGCS y FOGCH con 10 iteraciones, para el par de imágenes "sphere".*

por lo tanto se obtiene el resultado en menor tiempo. Además está claro que es por la arquitectura utilizada, del tipo flujo de datos.

Del mismo modo que en las dos secuencias anteriores, en caso de que se realice la modificación al algoritmo original de Horn con la finalidad de identificar fácilmente los objetos que están en movimiento, es necesario partir de la información obtenida mostrada en las imágenes de la figura 8.13. Haciendo $||\nabla I|| \geq 5$ es posible obtener unas nuevas imágenes de vectores de flujo óptico, mostradas en la figura 8.14.

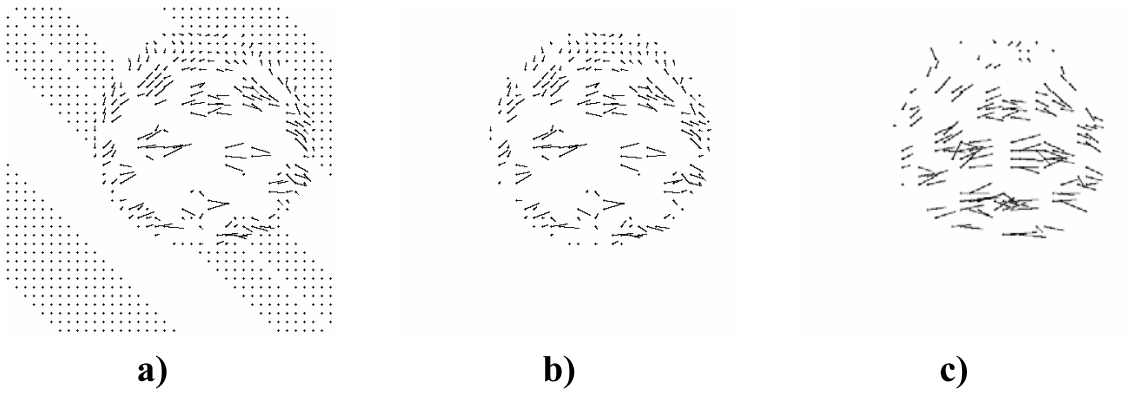


Figura 8.14: *Flujo óptico obtenido con un $t_h \geq 1$, 10 iteraciones y un $||\nabla I|| \geq 5$: a) flujo óptico procesando todos los píxeles (FOFP), b) flujo óptico guiado por cambios software (FOGCS) y c) flujo óptico guiado por cambios hardware (FOGCH).*

Observando las imágenes de la figura 8.14, de la misma forma que en las dos secuencia anteriores, sólo se obtiene una mejora en las imágenes a) y b). La razón es la misma expuesta en el análisis de las secuencias *taxi* y *Rubic*. Sin embargo, en este caso y debido a que el tipo de secuencia es sintética es innecesario utilizar técnica alguna, para mejorar la visualización de la imagen de vectores de flujo óptico. De hecho es posible observar que incluso la respuesta del sistema es mejor, en todos los aspectos, al utilizar una secuencia del tipo sintética. Por ejemplo, en la imagen a) de la figura 8.14, se muestra una cantidad de vectores de flujo óptico o puntos dentro de la imagen resultante que son debidos al fondo de la imagen y no tienen nada que

ver con el movimiento. La razón es el utilizar la información de los gradientes, lo que genera esa información errónea. En cambio la arquitectura que se está evaluando, que utiliza la técnica de procesado guiado por cambios, en ningún caso presenta algún vector de flujo óptico o información debida al fondo de la imagen.

Por otro lado, al utilizar este tipo de imágenes donde en cierta manera se controla la intensidad o brillo y todo tipo de parámetros que pudiese introducir ruido, se ayuda a disminuir considerablemente el porcentaje de píxeles a procesar. En este caso el porcentaje de píxeles procesados es del 27,17 %. Adicionalmente a esto, es innecesario una modificación del umbral t_h, para la detección de cambios, siendo suficiente el establecido con un $t_h \geq 1$. Y con este umbral es posible procesar 77,5 fps. En este caso, a diferencia de las secuencias de imágenes previamente evaluadas *taxi* y *Rubic*, la frecuencia de procesado es la misma en cualquier par de imágenes procesadas. Esto es debido a que es una secuencia sintética, a que el movimiento entre par de imágenes consecutivas es constante y que no existe ruido dentro de la secuencia de imágenes.

8.2.1. Aumento del rendimiento

Partiendo de los resultados anteriores es posible concluir que el aumento del rendimiento, con un $t_h \geq 1$, presenta un incremento mayor cuando se procesan imágenes sintéticas, como es el caso de la secuencia *sphere*. Sin embargo, cuando se procesan secuencias de imágenes naturales existe un incremento del rendimiento razonable. Dentro del mismo tipo de secuencias naturales, es mayor el incremento del rendimiento cuando son secuencias de imágenes capturadas en ambientes controlados, como es el caso de la secuencia *Rubic*. En el caso de procesar la secuencia *taxi* existe un incremento del rendimiento, aunque reducido, a pesar de que ese tipo de secuencia haya sido capturada en el exterior o bien en un ambiente no controlado. Lo antes dicho es posible observarlo en la gráfica mostrada en la figura 8.15.

La gráfica de la figura 8.15 emplea en todos los casos un umbral $t_h \geq 1$, a pesar de que las secuencias son de características distintas. Esto quiere decir que una secuencia natural no debe utilizar el mismo umbral que una secuencia sintética, debido a que esta última secuencia en ningún momento presentará ruido debido al medio ambiente o a la cámara, como en el caso de secuencias naturales. Por tal razón en el caso de procesar las secuencias de imágenes naturales es necesario incrementar el valor del umbral t_h, con el objetivo de eliminar ruido de la escena y tener un resultado visual aceptable. De esta manera si la arquitectura diseñada se

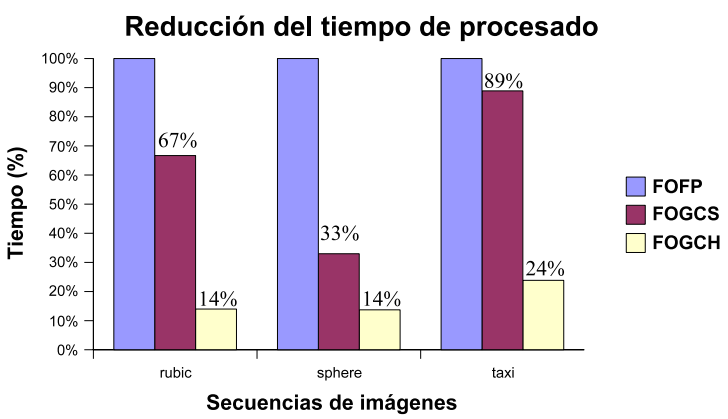


Figura 8.15: *Comparación de tiempos de procesado, para el cálculo del flujo óptico, en el cual se muestra la reducción del tiempo logrado con la arquitectura diseñada que emplea la estrategia del procesado guiado por cambios, utilizando un* $t_h \geq 1$.

ajusta con un umbral adecuado, en cada una de las secuencias de imágenes naturales analizadas y se compara la respuesta con la obtenida por el algoritmo de Horn, con un $||\nabla I|| \geq 5$, es posible ver que se obtiene una respuesta visual aceptable, como se puede apreciar en la figura 8.16.

Las imágenes a), b) y c) son las procesadas mediante la técnica FOFP y las imágenes a'), b') y c') son las procesadas mediante la técnica FOGCH. Es posible apreciar que las imágenes de flujo óptico obtenidas mediante la técnica FOGCH presentan una mayor inmunidad al ruido. Adicionalmente a eso se obtiene una reducción en los tiempos de procesado, los cuales son mostrados en la gráfica de la figura 8.17. Con la gráfica es posible concluir que el aumento del rendimiento se incrementa considerablemente. Por otro lado aunque en el procesado FOFP es posible modificar el umbral $||\nabla I|| \geq 5$, ya sea incrementándolo o disminuyéndolo, los tiempos de procesado se mantendrán constantes puesto que se procesan siempre todos los píxeles de las imágenes sin importar el valor del umbral $||\nabla I||$.

Por todo esto es se entiende que, para la arquitectura FOGCH, el incremento del rendimiento depende directamente del número de píxeles que cambiaron. Ahora bien, el número de píxeles que cambian depende de varios factores como son:

1. El tipo de secuencia.
2. Tamaño de las imágenes.
3. Medio ambiente, donde son capturadas las imágenes.
4. Sensibilidad de la cámara.

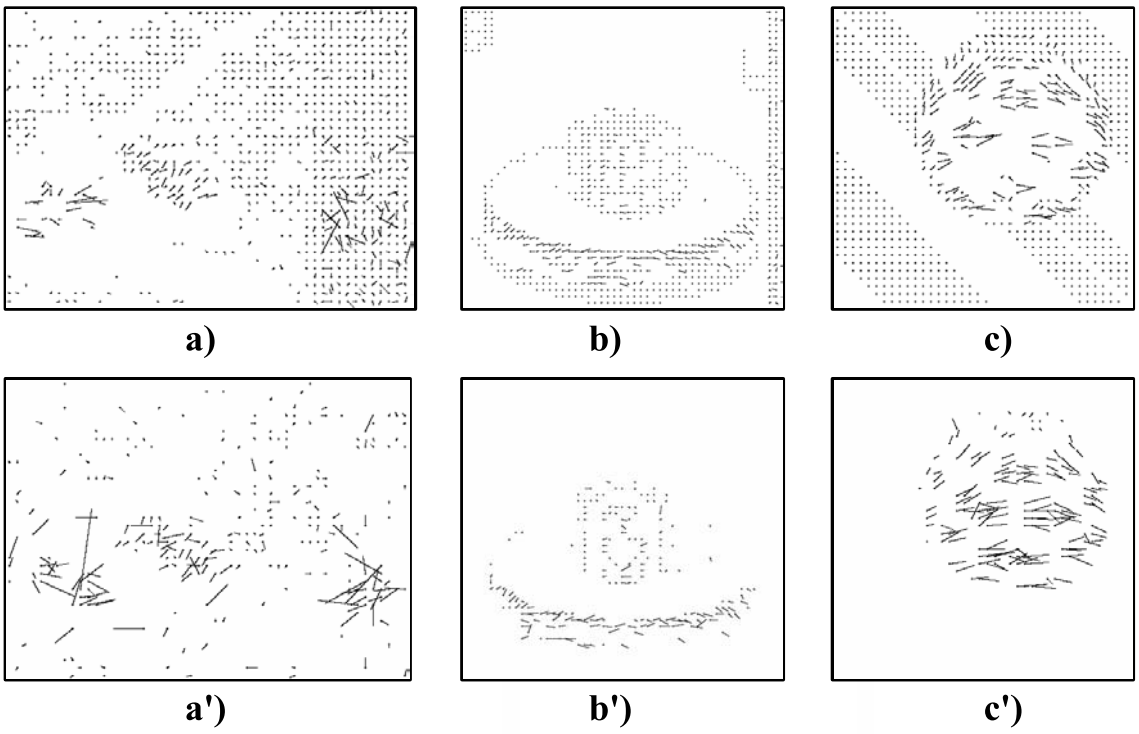


Figura 8.16: *Comparación visual de los resultados obtenidos por el algoritmo de Horn modificado (FOFP) y la técnica de procesado guiado por cambios utilizando la arquitectura diseñada (FOGCH) y umbrales adecuados para el tipo de secuencia.*

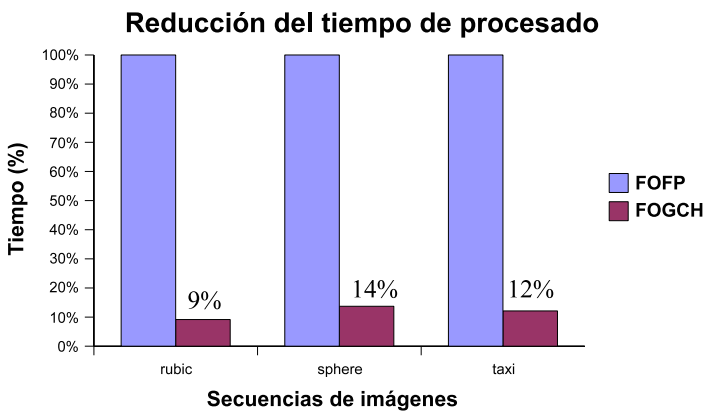


Figura 8.17: *Aumento del rendimiento con la estrategia de procesado guiado por cambios implementada en la arquitectura diseñada.*

5. Frecuencia de captura de las imágenes.
6. Valor del umbral (t_h).

Secuencia	**Tamaño de imagen**	**No. fps**	$t_h \geq$
Taxi	190×256	91,7	5
Rubic	240×256	77,3	3
Sphere	200×200	77,5	1

Cuadro 8.6: *Secuencias analizadas que muestra el número de imágenes procesadas por segundo, en función de un umbral adecuado para cada secuencia, utilizando la arquitectura FOGCH, con 10 iteraciones.*

Todos los factores se reflejan directamente en el número de píxeles que cambian entre pares de imágenes consecutivos, por esta razón, no es posible definir un valor exacto del número de imágenes que son procesadas en la arquitectura FOGCH. Sin embargo, en función del cuadro 8.6, en el que se presentan las tres secuencias analizadas, es posible plantear de forma empírica el número de imágenes que es posible procesar en la arquitectura. En el cuadro se indica la secuencia, el tamaño de las imágenes, número de imágenes procesadas y el umbral utilizado.

Entre mayor control se tenga durante la captura de las secuencias de imágenes, como es el caso de una secuencia sintética, el umbral utilizado es menor. En caso opuesto, entre menor control se tenga durante la captura de las imágenes, el umbral utilizado deberá ser mayor, pero hasta un cierto valor. De esta manera se procesan el mismo número de imágenes de la secuencia *sphere* que de la *Rubic*, con umbrales $t_h \geq 1$ y $t_h \geq 3$ respectivamente, a pesar de que las imágenes, que constituyen a la secuencia *Rubic*, son de un tamaño mayor que las de la secuencia *sphere*. De hecho en la secuencia *taxi* se procesan hasta 91,7 fps muy por arriba del número de imágenes procesadas en la secuencia *sphere*, que son de 77,5 fps, a pesar de que son de mayor tamaño las imágenes de la secuencia *taxi*. Si se promedian dichos valores, sólo como una aproximación empírica, se alcanzan a procesar aproximadamente 80 fps en la arquitectura diseñada.

Si comparamos el rendimiento obtenido en este trabajo con otras arquitecturas existentes, mostradas en el cuadro 8.7, se puede observar que el resultados del trabajo supera considerablemente a los trabajos ya propuestos. En todos los casos se utilizan el algoritmo iterativo de Horn y Schunck.

Es importante resaltar que el número de iteraciones realizadas en cada arquitectura es diferente. Entre mayor sea el número de iteraciones disminuye el error del flujo óptico obtenido para ese par de imágenes procesadas. En el caso del trabajo de Martín et al. [MZC+05] que realiza una sola iteración este conserva el flujo óptico

Trabajo:	Tamaño de imagen (píxeles)	No. fps	No. iter./par de imágenes
Cobos et al. [CM03]	50×50	19	3
Martín et al. [MZC+05]	256×256	60	1
Aquí propuesto	256×256	≈ 80	10

Cuadro 8.7: *Comparación con otras arquitecturas que implementan el algoritmo de Horn y Schunck.*

resultante. Los datos son utilizando como entrada para el procesado de un nuevo par de imágenes. Este proceso lo hace durante 60 imágenes consecutivas y en la imagen 61 inicia el proceso nuevamente. En el caso del trabajo de Cobos et al. [CM03] sólo se realizan 3 iteraciones. En el trabajo aquí propuesto se realizan 10 iteraciones pero sólo de los píxeles que cambiaron o de las componentes de dichos píxeles.

8.3. Precisión

Para poder efectuar una evaluación de la precisión del sistema diseñado, es necesario llevar a cabo una evaluación formal de los resultados. Por esto es necesario conocer el flujo óptico real existente en la escena. También es necesario conocer el flujo óptico obtenido utilizando el algoritmo original de Horn, que procesa todos los píxeles de las imágenes, de esta manera es posible evaluar relativamente el resultado obtenido con la arquitectura diseñada. La razón es que la arquitectura diseñada utiliza el algoritmo de Horn con una variante que consiste en el procesado guiado por cambios. De esta manera se obtienen dos tipos de errores en la evaluación: El primer error es la diferencia entre el flujo óptico real y el flujo óptico obtenido, mediante la arquitectura diseñada. El segundo tipo es el error de referencia, y es el que se obtiene mediante la diferencia entre el flujo óptico obtenido con la arquitectura diseñada y el flujo óptico obtenido utilizando la estrategia original de Horn (FOFP), el cual procesa todos los píxeles de las imágenes.

El error se puede expresar como un error absoluto o una relación señal a ruido entre el movimiento detectado por cualquier algoritmo utilizado para el cálculo del flujo óptico y el flujo óptico real existente.

En este caso se calcula el error angular entre el flujo óptico real y el flujo óptico estimado, representado por la ecuación 8.1.

FOFP		**FOGCS**			**FOGCH**		
Error	**Dev. St.**	**Error**	**Err. R**	**Dev. St.**	**Error**	**Err. R**	**Dev. St.**
$55,228°$	$11,189°$	$57,42°$	$2,19°$	$10,44°$	$57,60°$	$2,37°$	$9,90°$

Cuadro 8.8: *Error medio y desviación estándar de FOFP, error medio, error de referencia y desviación estándar de FOGCS y FOGCH. Cálculos realizados al par de imágenes "Square2", con un $t_h \geq 1$ y 10 iteraciones.*

$$\Psi_E = arccos(\frac{u_c u_e + v_c v_e + 1}{\sqrt{(u_c^2 + v_c^2 + 1)(u_e^2 + v_e^2 + 1)}}) \tag{8.1}$$

donde (u_c, v_c) representa el flujo óptico correcto y (u_e, v_e) representa el flujo óptico estimado. Con la finalidad de manejar medidas de errores que son utilizadas comúnmente en la bibliografía, es posible utilizar la ecuación 8.1 para estimar el error medio y la desviación estándar.

Para conocer el flujo óptico correcto, es necesario contar con secuencias de imágenes que cumplan con esta característica. De esta manera se utilizan secuencias de imágenes sintéticas de las cuales se conoce el flujo óptico real. Las secuencias de imágenes utilizadas para este propósito fueron 3, comúnmente utilizadas en la literatura: *treed*(*Diverging tree*), *treet*(*Traslating tree*) y *Square2*.

De la misma forma que en la evaluación visual, aquí se realiza el cálculo del flujo óptico mediante las estrategias FOFP, FOGCS y FOGCH. Pero, adicionalmente se calcula el error medio, el error relativo y la desviación estándar, para cada una de las secuencias. En todos los casos se utilizó un $t_h \geq 1$ y 10 iteraciones.

Secuencia *Square2*

La secuencia de imágenes *Square2* es la primera en ser analizada. Esta secuencia consiste en una traslación de un cuadrado hacia la esquina superior derecha de la imagen. La imagen original, del cuadrado, se muestra en la figura 8.18. El cuadrado es de tamaño de 10×10 píxeles y se desplaza a una velocidad constante de $(\frac{4}{3}, \frac{4}{3})$. Debido a que en esta sección el interés radica exclusivamente en realizar un análisis formal se calcula el error medio, el error relativo y la desviación estándar, los cuales se muestran en el cuadro 8.8. Sin embargo, también se presentan las imágenes de flujo óptico obtenidas mediante FOFP y FOGCH, como se muestra en la figura 8.18.

Es posible observar que el error medio es grande en todos los casos, incluso medi-

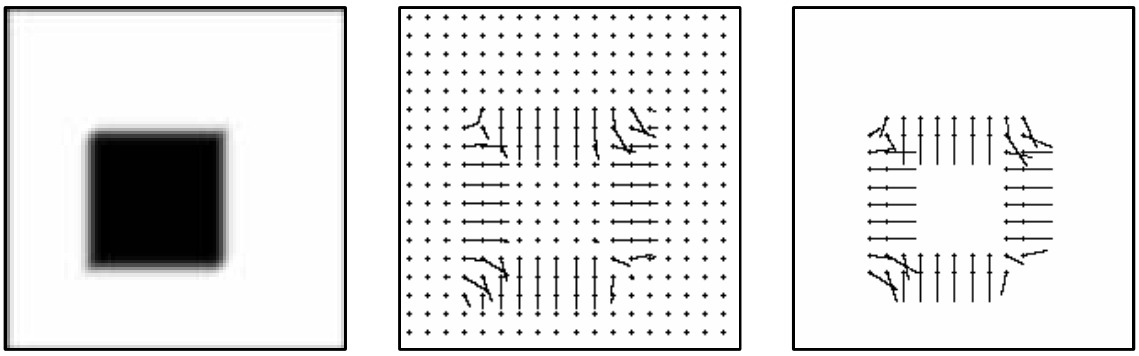


Figura 8.18: *Imagen original de la secuencia Square2 y el flujo óptico calculado mediante FOFP y FOGCH.*

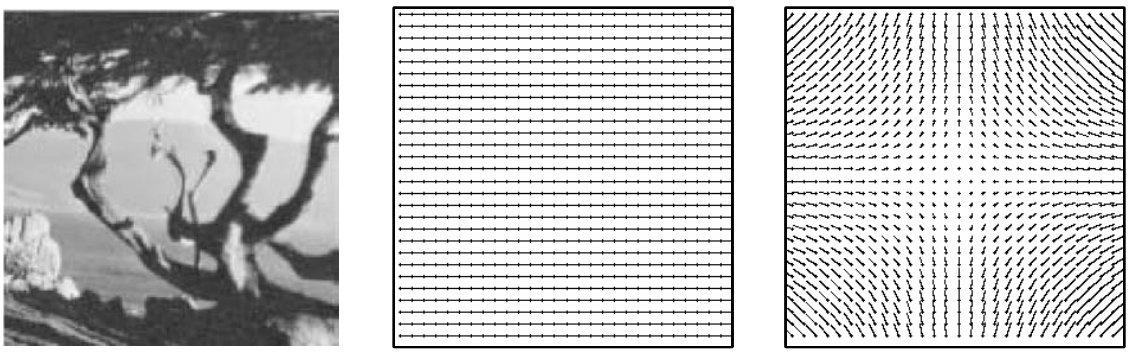
ante la técnica FOFP. Al utilizar el procesado del FOGCH se observa un incremento en el error medio de sólo $2,37°$. Sin embargo, en la desviación estándar se obtiene una mejora, reduciendo de $11,189°$ a $9,9°$. En la figura 8.18, donde se muestran los vectores de flujo óptico calculados, se observa que dichas diferencias, en el error medio y la desviación estándar, apenas son identificables.

Secuencias *treet* y *treed*

A continuación se realiza el análisis de las secuencias *treet* y *treed*. Estas secuencias son más complejas que la secuencia *Square2*. En las secuencias se simula un desplazamiento de la cámara respecto a una escena estática, la cual es mostrada en la figura 8.19 a). En la secuencia *treed* se simula un movimiento de la cámara hacia el centro de la escena; el foco de expansión se desplaza hacia el centro de la imagen a una velocidad de 1,29 píxeles/frame del lado izquierdo y a 1,86 píxeles/frame del lado derecho. Por otro lado, en la secuencia *treet* se simula un movimiento horizontal, hacia la derecha, de la cámara respecto a la escena, con una velocidad de 1,73 a 2,26 píxeles/frame. La escena estática y los respectivos campos de flujo óptico correctos se muestran en la figura 8.19.

De la misma manera que en los análisis anteriores, para cada una de las secuencias de imágenes se realiza el procesado del FOFP, FOGCS y FOGCH. En cada proceso se calcula el error medio y la desviación estándar. Adicionalmente al procesar el FOGCS y FOGCH se presenta el error de referencia.

Para el caso de la secuencia *treed* se obtienen los valores mostrados en el cuadro 8.9. Del cuadro se puede observar que al procesar la información utilizando la técnica



a) **b)** **c)**

Figura 8.19: *a) la imagen original de la escena, b) el flujo óptico correcto de la secuencia "treet " y c) el flujo óptico correcto de la secuencia "treed".*

FOFP		**FOGCS**			**FOGCH**		
Error	**Dev. St.**	**Error**	**Err. R**	**Dev. St.**	**Error**	**Err. R**	**Dev. St.**
$13,56°$	$11,79°$	$18,01°$	$4,45°$	$14,26°$	$21,62°$	$8,06°$	$13,80°$

Cuadro 8.9: *Error medio y desviación estándar de FOFP, error medio, error de referencia y desviación estándar de FOGCS y FOGCH. Cálculos realizados al par de imágenes "treed", con un $t_h \geq 1$ y 10 iteraciones.*

del FOGCH el error promedio se incrementa en un $8,06°$ respecto al error obtenido al utilizar la técnica de FOFP, que utiliza el algoritmo original de Horn y aritmética de coma flotante. Adicionalmente la desviación estándar también se incrementa, pero sólo un $2,01°$. Al calcular el FOGCS, que utiliza aritmética de coma flotante, se aprecia que el error se incrementa sólo $4,45°$, casi la mitad de lo que se incremento al realizar el cálculo mediante FOGCH.

Para el caso de la secuencia *treet* se obtienen los valores mostrados en el cuadro 8.10. En el cuadro se aprecia que el error medio es elevado en todos los cálculos, pues ronda los $41°$. Al realizar el cálculo mediante el FPGCH el error medio se incrementa un $3,88°$ y la desviación estándar se mejora un $1,73°$, todo esto respecto al flujo óptico calculado utilizando el algoritmo original de Horn y procesando todos los píxeles de las imágenes en aritmética de coma flotante (FOFP). En el caso del procesado FPGCS se considera, en función de sus resultados, que es un punto intermedio entre los resultados obtenidos por FOFP y FOGCH. Probablemente la razón es que a pesar de que emplea la técnica del procesado guiado por cambios también utiliza aritmética de coma flotante lo que ayuda a obtener mejores resultados.

FOFP		**FOGCS**			**FOGCH**		
Error	**Dev. St.**	**Error**	**Err. R**	**Dev. St.**	**Error**	**Err. R**	**Dev. St.**
$41,04°$	$26,92°$	$43,92°$	$2,88°$	$26,32°$	$44,92°$	$3,88°$	$25,19°$

Cuadro 8.10: *Error medio y desviación estándar de FOFP, error medio, error de referencia y desviación estándar de FOGCS y FOGCH. Cálculos realizados al par de imágenes "treet", con un* $t_h \geq 1$ *y 10 iteraciones.*

Parte IV

Conclusiones

Capítulo 9

Conclusiones y trabajos futuros

En este capítulo se resume el trabajo de investigación realizado, se enumeran las aportaciones que se han efectuado al estado de la investigación, y se plantean las futuras líneas de trabajo como continuación de la investigación presentada.

9.1. Sumario

9.1.1. Origen y planteamiento del proyecto de investigación

El origen del presente trabajo de investigación parte de la confluencia de varias líneas de trabajo en el seno del grupo de Tecnologías y Arquitecturas de la PErcepción por Computador (TAPEC), grupo de investigación y desarrollo perteneciente al Departamento de Informática de la Escuela Técnica Superior de Ingeniería y al Instituto de Robótica, de la Universidad de Valencia[1]. En el grupo de trabajo se tiene experiencia en varias áreas, tales como percepción por computador, lenguajes de descripción de hardware, inteligencia artificial y arquitectura de computadores. El trabajo surge del estudio de varios trabajos de investigación, previamente desarrollados, los cuales fueron aportando ideas para la definición del presente trabajo de tesis doctoral. El trabajo de investigación, que realiza la primera aportación para la definición del presente trabajo, fue el desarrollado en el proyecto del MCYT, titulado: *Análisis de movimiento de alta velocidad mediante el uso de imágenes foveales y visión binocular entrelazada* (TIC2001-3546). En este trabajo se estudian arquitecturas, algoritmos y el uso de imágenes foveales. El segundo trabajo, titulado:

[1] http://tapec.uv.es

Técnicas empotradas para el procesado en tiempo real de la información perceptual de robots móviles (UV-AE-20050206), aborda técnicas para el procesado en tiempo real y estudios para la percepción visual. Estos últimos estudios condujeron a una propuesta novedosa y poco utilizada en el área de la visión por computador. De esta manera surge una nueva propuesta de investigación con el proyecto titulado: *Desarrollo de técnicas de análisis de secuencias de imágenes basadas en procesado por cambios y su aplicación al reciclado automático* (UV-AE-20060242). Así también, el proyecto aceptado recientemente por el MCYT, titulado: *Desarrollo de técnicas y sensor para visión asíncrona guiada por cambios para el análisis de movimiento a muy alta velocidad* (TEC2006-08130/MIC). Así, con los proyectos desarrollados previamente surge el presente trabajo de investigación.

En el área de la percepción por computador existen distintas técnicas de sensorización entre las que destaca la sensorización visual, la cual permite conseguir simultáneamente un largo alcance y una precisión muy aceptable. Sin embargo, como desventaja de los métodos de sensorización visuales habituales, se tiene la gran cantidad de información a procesar, la complejidad y el coste computacional de muchos algoritmos de visión artificial. Por otro lado, en muchos casos es necesaria una respuesta en tiempo real, por lo tanto se hace deseable una solución con *hardware* específico para conseguir dicho propósito. Por tal razón es recomendable realizar diseños a la medida pero también es necesario o deseable una cierta flexibilidad *software*.

Los dispositivos lógicos programables combinan simultáneamente la velocidad del *hardware* con una cierta programabilidad, por lo que se ha planteado la utilización de un sistema basado en lógica reconfigurable para el procesamiento de la información visual. La estrategia para el procesado de la información, siempre que sea posible, utilizará una arquitectura que procese la información en paralelo.

Existen una gran variedad de arquitecturas para el procesamiento de imágenes en tiempo real. Una revisión del estado del arte de las arquitecturas existentes ha tenido una utilidad limitada, pues no se ha encontrado una arquitectura específica que se ajuste a las necesidades de procesamiento del algoritmo utilizado en este trabajo. Si bien el algoritmo ya ha sido implementado con ciertas arquitecturas, el hecho de utilizar la técnica propuesta de procesado guiado por cambios, marca un cambio respecto a otras arquitecturas. Por otro lado, surge la necesidad de incorporar un módulo de detección de cambios, que permita seleccionar adecuadamente la información relevante y útil para el sistema de visión. Esto aumenta en cierto grado la complejidad de la arquitectura de procesado a desarrollar.

El uso del flujo óptico surge como una de las mejores alternativas para la medición del movimiento en el espacio bidimensional, pues es tan importante para un sistema de visión artificial como para un organismo vivo. Sin embargo, los algoritmos que realizan el cálculo del flujo óptico son muy complejos y tienen un costo computacional muy elevado. Muchos trabajos desarrollados que emplean el cálculo del flujo óptico, presentan el tiempo de cálculo como la principal deficiencia e incluso sólo se abocan en la precisión de los resultados. Sin embargo, existe una gran cantidad de técnicas para realizar el cálculo del flujo óptico. Cada técnica puede ser más precisa que otra de acuerdo al tipo de secuencia de imágenes que son procesadas.

Una característica observada en todos los sistemas artificiales de visión es que realizan el procesado de la información de todos los píxeles que intervienen en las imágenes. Este hecho es totalmente diferente al que realizan los sistemas de visión biológicos, donde el envío de la información visual por parte del ojo humano al cerebro se realiza de forma asíncrona, por parte de cada píxel de forma individual; sin embargo, prácticamente toda la visión artificial actual está basada en la adquisición síncrona de imágenes completas. La principal ventaja del modelo biológico es la capacidad de poder reaccionar ante estímulos en el instante mismo en que se producen y no a intervalos fijos preestablecidos como en el modelo clásico de tratamiento de imágenes.

Así pues, la utilización del flujo óptico y su implementación en dispositivos reconfigurables es una alternativa para realizar e implementar una arquitectura para el análisis del movimiento en imágenes a alta velocidad. Por otro lado, la complejidad de los algoritmos para el cálculo del flujo óptico y la cantidad de datos que posee una imagen, hace necesario realizar un procesado asíncrono de la información. A este tipo de procesado se le denomina procesado guiado por cambios.

9.1.2. Diseño de la arquitectura

El estudio de los algoritmos a implementar ha definido las características de la arquitectura de cada módulo diseñado. Básicamente la arquitectura parte de los algoritmos que emplea Horn y Schunck para el cálculo del flujo óptico y algunas modificaciones para llevar a cabo su implementación en la FPGA utilizada. Además se incorporan nuevos bloques que realizan procesos adicionales, pero de forma concurrente, para emplear la técnica de procesado guiado por cambios. De esta manera se deben considerar varios puntos específicos en el diseño de la arquitectura.

Primero, el algoritmo de Horn y Schunck emplea básicamente funciones que utilizan operaciones de suma, resta, division y multiplicación.

Segundo, el módulo para la detección de los cambios existentes en dos imágenes consecutivas, es implementado con una operación de resta y una operación que obtiene su valor absoluto, el cual se compara con un umbral establecido previamente.

Finalmente la lógica de control es establecida por la arquitectura utilizada, específicamente por la forma en como son manejados los datos. En este caso es un control distribuido. Si bien la arquitectura es síncrona los datos son tratados de manera asíncrona. Esto es debido a que son procesados sólo aquellos píxeles que presentan cambios significativos entre dos imágenes consecutivas, con un intervalo de muestreo muy pequeño. De esta manera, la lógica de control maneja los datos a procesar en función de los propios datos. La arquitectura que se maneja así es llamada arquitectura de flujo de datos. Debido a la forma cómo se manejan los datos, es necesario dividir la arquitectura en varios módulos, y cada uno de ellos puede estar segmentado o no. Cada módulo posee una memoria local, tanto de entrada como de salida y cuenta con un circuito que detecta la disponibilidad de los datos.

Una vez hecho el análisis global de cómo se maneja la información, es posible exponer el diseño del primer módulo o etapa del sistema. Este módulo obtiene los gradientes y la tabla de detección de cambios llamada LUT. El módulo posee una memoria de entrada y una memoria para almacenar sus datos de salida. Las operaciones que realiza este bloque básicamente son de suma y resta, también una división por 4 que es implementada mediante dos desplazamientos. De esta manera, sólo es necesario un ciclo de reloj para procesar la información y es utilizado un diseño puramente combinacional. El circuito que detecta la disponibilidad de los datos es la misma memoria de entrada, la cual indica con las señales *full_fifo_grad_in* o *empty_fifo_grad_in* si la memoria está llena o vacía respectivamente. La memoria de entrada de este módulo recibe los datos directamente de la memoria SDRAM. Por otro lado la organización de la memoria de entrada está hecha de tal manera que puede almacenar dos renglones de cada imagen durante la lectura. Para el procesado de estos 4 renglones, el segundo renglón leído ocupará la posición del primer renglón leído y el espació vacío, dejado por el segundo renglón, será ocupado con un nuevo renglón (tercer renglón) de la imagen a ser procesada. En la memoria de salida se almacenan los gradientes y la tabla de cambios (LUT) la cual registra la posición de los píxeles que presentaron cambios significativos, por encima de un umbral previamente establecido.

El segundo módulo de la arquitectura realiza una selección de los píxeles que serán procesados en función de la tabla de cambios (LUT). Los gradientes y las laplacianas correspondientes a los píxeles seleccionados son almacenados en la memoria de entrada del tercer módulo de la arquitectura, el cual calcula la velocidad. El valor de las laplacianas es cero en la primera iteración. Por esta razón es necesario incorporar un circuito que realice el conteo del número de iteraciones efectuadas.

El tercer módulo se encarga de leer la memoria que contiene los gradientes y las laplacianas de los píxeles que cambiaron, y en base a ellos calcula las componentes de velocidad, las cuales son almacenadas en una memoria. Para realizar el cálculo de las componentes de la velocidad se utilizan varias operaciones de suma, multiplicación y dos divisiones. Todas las operaciones de suma y multiplicación son ejecutadas de manera combinacional y sólo en el caso de la división fue necesario segmentar la operación. Por tal razón el módulo presenta una latencia de 8 ciclos de reloj.

Posteriormente las componentes de velocidad son leídas para efectuar una reconstrucción que incluya las componentes de los píxeles que cambiaron. Esta reconstrucción se lleva acabo utilizando la tabla de cambios LUT y una vez efectuada la reconstrucción, las componentes de velocidad correspondientes a tres renglones son almacenadas en memoria.

El último módulo `Ctl_Lapla` realiza la lectura de las componentes de velocidad, las cuales son procesadas por el módulo que obtiene las componentes de la laplaciana. Estas componentes son almacenadas en una memoria intermedia, llamada memoria `fifo_lapla_all` para posteriormente seleccionar únicamente las laplacianas correspondientes a los píxeles que cambiaron, de acuerdo a la LUT. Los datos resultantes son almacenados en una memoria de salida junto con las componentes de los gradientes que pertenecen a los píxeles que presentaron cambios.

Para las siguientes iteraciones el segundo módulo o módulo de control `Ctl_Grad_Lapla` lee la memoria de salida del módulo `Ctl_Lapla` con el fin de procesar sus componentes y efectuar nuevamente el cálculo de velocidades.

Una vez concluido el número de iteraciones establecido, las velocidades obtenidas son almacenadas en la memoria SDRAM junto con la tabla de cambios o LUT.

Una característica adicional no comentada anteriormente es que en la arquitectura se utilizan dos relojes, uno que trabaja a una frecuencia de 33 MHz y otro que trabaja a 166 MHz. Esto se hizo para agilizar los procesos de escritura o lectura en algunas etapas del sistema que permiten trabajar a una frecuencia mayor. Tal es el caso de los procesos de lectura o escritura en la SDRAM y en las etapas intermedias

durante los procesos que requieren leer la tabla de cambios.

Adicionalmente a lo expuesto, se realizaron los bloques necesarios para la implementación y puesta a punto del sistema. Específicamente se trata de la comunicación entre el sistema desarrollado y la memoria SDRAM, y la comunicación entre el sistema completo y el bus PCI. El bus PCI se emplea como un estándar de interconexión entre los diferentes componentes que integran el sistema. De esta manera el controlador de la SDRAM y la interfaz PCI son necesarios para concluir el sistema desarrollado.

Finalmente con todos los módulos diseñados se implementa un sistema de visión para el análisis de movimiento a alta velocidad, empleando un algoritmo para el cálculo del flujo óptico y la estrategia de procesado guiado por cambios. Los cambios son aquellos detectados dentro de una secuencia de imágenes.

9.1.3. Implementación física y resultados

Se ha diseñado un sistema capaz de realizar el cálculo del flujo óptico en tiempo real, utilizando la técnica de procesado guiado por cambios. El sistema fue realizado e implementado en una tarjeta de desarrollo Stratix PCI la cual posee características muy flexibles, entre las que destacan:

- Dispositivo Principal: FPGA Stratix, EP1S60F1020C6.
- Memoria de 256 MBytes. PC333 DDR SDRAM (SODIMM).
- Memoria Flash con capacidad de 64 Mbits.
- Interfaz para programación USB Blaster II.
- Interfaz del Bus PCI, compatible con bus de 32 ó 64 bits y a 3,3 V o 5 V.

El componente principal es la FPGA, Stratix EP1S60F1020C6, la cual posee las características mostradas en el cuadro 9.1.

Una vez sintetizado el programa se realiza la configuración de la FPGA mediante la interfaz JTAG. Esto se hace una vez encendida la tarjeta de desarrollo Stratix PCI. La interfaz JTAG permite al software Quartus II descargar el archivo (**stratix_top.sof**) en la FPGA, utilizando el cable USB Blaster II. Una vez configurada la FPGA se reinicia el sistema. Al reiniciarse la computadora, el sistema operativo detecta un nuevo dispositivo hardware y por lo tanto el sistema automáticamente ejecuta el software para seleccionar e instalar el controlador apropiado. El controlador fue diseñado previamente utilizando la herramienta de desarrollo Jungo

Característica	**No. componentes**
LEs	57.120
M512 RAM (32 × 18 bits)	574
M4K RAM (128 × 36 bits)	292
M-RAM (4K × 144 bits)	6
TOTAL bits de RAM	5.215.104
DSPs	18
Multiplicadores embebidos	144
PLLs	12
Terminales de I/O	1022

Cuadro 9.1: *Características de la FPGA EP1S60F1020C6, de la familia Stratix de Altera.*

WinDriver[2]. La simplicidad de este software de desarrollo hace posible diseñar un controlador en poco tiempo.

Una vez que se cuenta con el controlador es posible ejecutar el programa asociado con el hardware desarrollado. El programa actúa como Maestro e inicia el procesado de la información, en este caso el procesado de dos imágenes consecutivas. También permite obtener información relevante como el campo de flujo óptico, tiempos de procesado y magnitud del error, de tal manera que es posible realizar una evaluación visual y determinar el aumento del rendimiento y la precisión del sistema desarrollado.

Para realizar la evaluación del sistema se utilizaron distintas clases de secuencias de imágenes. La primera clase es de tipo natural y la segunda clase es de tipo sintética. Dentro de las secuencias naturales están las que son capturadas en un ambiente controlado, en referencia a los cambios de iluminación, como la secuencia *Rubic*, y las que son capturadas en ambientes abiertos, sin ningún tipo de control en los cambios de iluminación como la secuencia *taxi*. Por otro lado, dentro de las imágenes sintéticas están las secuencias que desean evaluar algún fenómeno como las transparencias en el caso de la secuencia *sphere*, o movimientos traslacionales o divergentes como las secuencias *tree*y *treed*, respectivamente.

Al evaluar la arquitectura se llegaron a varias conclusiones importantes:

1. Se evaluaron tres secuencias de imágenes, con un mismo umbral $t_h \geq 1$, la

[2]http://www.jungo.com

primera del tipo sintética (*sphere*), la segunda del tipo natural capturada en un ambiente controlado (*Rubic*) y la tercera, también del tipo natural pero capturada en el exterior (*taxi*). De las evaluaciones se concluyó que el rendimiento es mucho mayor en las secuencias de imágenes sintéticas, pues ronda el 85 %, como es el caso de la secuencia *sphere*. Para las imágenes naturales el rendimiento disminuye cuando no existe control en el medio durante la captura de las imágenes, como es la secuencia *taxi* que presentó el peor rendimiento, el cual ronda el 75 %.

2. En el caso de trabajar con secuencias naturales, se debe realizar un incremento en el valor del umbral. El objetivo de incrementar el umbral es útil para eliminar la mayor cantidad de ruido introducido por la cámara o por el medio ambiente al capturar la secuencia. La selección del umbral se hace en función de cómo y bajo que condiciones se realiza la adquisición de las imágenes. De esta manera, para la secuencia *taxi* el umbral establecido es mayor ($t_h \geq 5$) que para la secuencia *Rubic* el cual se estableció con un $t_h \geq 3$.
3. Si el umbral se ajusta a un valor adecuado para cada una de las secuencias *taxi* y *Rubic*, se obtiene un incremento del rendimiento, incluso mayor al registrado por la secuencia de imágenes sintéticas *sphere*.
4. Para la evaluación de la precisión es necesario utilizar sólo imágenes sintéticas, obteniéndose el error medio, el error de referencia y la desviación estándar. En dicha evaluación se concluye que el error de referencia es aceptable, comparado con el obtenido por un sistema que utilice el algoritmo original de Horn, en el cual se procesan todos los píxeles de las imágenes y se realizan las operaciones en coma flotante. Se considera el error aceptable debido a que en el peor de los casos el error de referencia es de $8°$ y en el mejor es de $2,37°$. Por otro lado la desviación estándar es incluso mejor, en algunos casos, que el valor obtenido por el algoritmo original de Horn. Por ejemplo, para la secuencia *Square2* disminuye de $11,189°$ a $9,9°$ y en la secuencia *treet* disminuye de $26,92°$ a $25,19°$. El peor de los casos se obtiene con la secuencia *treed* que se incrementa de $11,79°$ a $13,8°$.
5. Muchos trabajos de visión que emplean el flujo óptico como medida de movimiento, se abocan específicamente a mejorar la precisión. Sin embargo, ninguno de ellos puede presentar el mejor resultado en todos los tipos de escenas. Es decir, un algoritmo específico para el cálculo del flujo óptico, puede ser más eficiente que otro algoritmo para un tipo de escena específico. Por lo que no existe un algoritmo que presente el comportamiento ideal para cualquier tipo de escena.

9.2. Aportaciones

A continuación se enumeran, en forma resumida, las aportaciones realizadas:

- Se ha propuesto una nueva técnica para el procesamiento de imágenes, la cual consiste en procesar sólo los píxeles que presentaron cambios significativos. Obteniéndose un aumento considerable en el rendimiento.
- La propuesta surge del análisis del sistema visual biológico. Los sistemas de visión biológicos envían la información visual de forma asíncrona, cada píxel de forma individual; sin embargo, prácticamente toda la visión artificial actual está basada en la adquisición síncrona de imágenes completas. La principal ventaja del modelo biológico es la capacidad de poder reaccionar ante estímulos en el instante mismo en que se producen y no a intervalos fijos preestablecidos como en el modelo clásico de tratamiento de imágenes, los cuales procesan todos los píxeles de manera consecutiva. De tal manera que la aportación hecha es proponer una arquitectura que intenta reproducir un comportamiento biológico.
- Actualmente en ningún trabajo en el que se haya calculado el flujo óptico, utilizando el algoritmo de Horn y Schunck, en tiempo real o utilizando arquitecturas dedicadas, proporciona resultados acerca del rendimiento del sistema es decir, de su exactitud y eficiencia. En este trabajo se evalúa realmente la magnitud del error al utilizar aritmética entera.
- Se ha realizado una arquitectura en lógica reconfigurable específica para el análisis del movimiento. Los resultados obtenidos combinan flexibilidad *software* y prestaciones *hardware*, y apuntan a que este es un excelente camino y campo de aplicación para las máquinas reconfigurables.
- Se ha diseñado una arquitectura orientada al cálculo del flujo óptico utilizando un procesamiento guiado por cambios. Algunos bloques se han diseñado para procesar la información en cauce segmentado, de esta manera se acelera la capacidad de cálculo. La arquitectura definida es independiente de la implementación realizada. De hecho, es posible implementar la misma arquitectura en un dispositivo HardCopy Stratix, programable en máscara, que posee las mismas características que su equivalente FPGAStratix, con una mejora en promedio del 50 % y con una reducción en el consumo de hasta un 40 %.
- La arquitectura implementada es del tipo flujo de datos; esta arquitectura no es convencional pues emplea memoria de entrada y de salida para cada módulo de procesamiento. Además, la lógica de control empleada es del tipo distribuida lo

que significa que las señales de control son independientes en cada módulo.

- Se ha implementado, para su utilización en una arquitectura reconfigurable, un algoritmo de detección de movimiento, independiente del movimiento de la cámara.
- Para la implementación de la arquitectura se seleccionó una plataforma flexible con una FPGA robusta que permite una fácil modificación o sustitución del algoritmo utilizado, haciendo al sistema más robusto. Por otro lado, se obtiene flexibilidad gracias a la interfaz PCI incorporada, la cual permite un prototipado rápido y prueba del sistema.

9.3. Trabajo futuro

El presente trabajo de investigación da pauta a numerosas líneas de trabajo futuro en diversos campos. Algunas de estas líneas serán abordadas directamente por el doctorando y por el grupo de trabajo TAPEC, grupo al cual pertenece. Otros trabajos, que pueden ser continuación del mismo, serán realizados conjuntamente con otros investigadores del Instituto Politécnico Nacional, en México.

Debido a que el sistema diseñado fue realizado utilizando una FPGA de última generación, la cual posee más de 5 millones de bits de memoria y 144 multiplicadores embebidos, entre otras características sobresalientes, hace que la continuación inmediata sea intentar realizar modificaciones a la arquitectura con la finalidad de hacerla más robusta y/o más precisa. Por ejemplo, es posible evaluar la posibilidad de realizar las operaciones en coma flotante, en particular la operación del cálculo de la velocidad. Otra alternativa con la intención de mejorar la respuesta, sería implementar un prefiltrado Gausiano espacio-temporal, como lo propone Barron et al. en [BFB94].

Si se quiere mejorar el rendimiento del sistema, es posible implementar otros algoritmos que permitan una paralelización de la arquitectura utilizando la técnica del procesado guiado por cambios. En este trabajo se implementó el algoritmo iterativo de Horn y Schunck, el cual limita la capacidad de paralelización de la arquitectura. En cambio existen otros algoritmos que presentan características satisfactorias e incluso en algunos casos mejores como el algoritmo propuesto por Lucas y Kanade.

Debido al aumento del rendimiento que se obtiene utilizando la estrategia de procesado guiado por cambios, es posible plantear una arquitectura alternativa para

aplicaciones específicas. Por ejemplo, en lugar de utilizar alguna técnica para calcular el flujo óptico, para el análisis del movimiento, es posible utilizar otra estrategia para el análisis del movimiento como podría ser la basada en la correspondencia.

Un último proyecto que continúa con el área de investigación desarrollada, es el trabajo aceptado por el MCYT titulado: *Desarrollo de técnicas y sensor para visión asíncrona guiada por cambios para el análisis de movimiento a muy alta velocidad* (TEC2006-08130/MIC). El trabajo consiste en desarrollar un sensor que proporcione asíncronamente los píxeles que serán procesados (sólo aquellos que presentaron cambios significativos) y de esta manera la información será procesada de forma asíncrona por una arquitectura hardware diseñada a la medida. Así se incrementa el rendimiento e incluso se ahorra área de la FPGA utilizada.

De manera adicional, se planteará el diseño de un conjunto de herramientas que automaticen el procesado de la información. En este trabajo sólo se desarrollaron programas que realizan una implementación fuera de línea, pero que son los necesarios para una conclusión final de la evaluación del trabajo desarrollado.

Parte V

Bibliografía

Bibliografía

[AA05] F. Ahmadianpour y M.O. Ahmad. A fast algorithm for motion estimation under the varying inter-frame brightness characteristics. *Journal of Electronic Imaging*, 14(4):43013–43020, December 2005.

[AB06] N. Ahuja y A. Briassouli. Joint spatial and frequency domain motion analysis. En *7th International Conference on Automatic Face and Gesture Recognition*, páginas 197–204, 2006.

[AC06] O.D. Arandjelovic y R. Cipolla. A new look at filtering techniques for illumination invariance in automatic face recognition. En *7th International Conference on Automatic Face and Gesture Recognition, FGR'06*, páginas 449–454, 2006.

[ALK07] T. Amiaz, E. Lubetzky, y N. Kiryati. Coarse to over-fine optical flow estimation. *Pattern Recognition, PR*, 40(9):2496–2503, September 2007.

[AMMS06] F. Archetti, C.E. Manfredotti, V. Messina, y D.G. Sorrenti. Foreground-to-ghost discrimination in single-difference pre-processing. En *Advanced Concepts for Intelligent Vision Systems, ACIVS'06*, páginas 263–274, 2006.

[Ana89] P. Anandan. A computational framework and an algorithm for the measurement of visual motion. *International Journal on Computer Vision*, 13(2):283–310, September 1989.

[Aya03] N. Ayache. Epidaure: A research project in medical image analysis, simulation, and robotics at inria. *IEEE Transactions on Medical Imaging*, 22(10):1185–1201, October 2003.

[BB95] S.S. Beauchemin y J.L. Barron. The computation of optical-flow. *ACM Computing Surveys*, 27(3):433–467, September 1995.

[BBF94] J.L. Barron, S.S. Beauchemin, y D.J. Fleet. On optical flow. En *Artificial Intelligence and Information-Control Systems of Robots, AI-ICSR'94*, páginas 3–14, 1994.

[BBF95] J.L. Barron, S.S. Beauchemin, y D.J. Fleet. On optical flow. *Artificial Intelligence*, 27(3):433–467, September 1995.

[BFB94] J.L. Barron, D.J. Fleet, y S.S. Beauchemin. Performance of optical flow techniques. *International Journal of Computer Vision*, 12(1):43–77, September 1994.

[BIMK04] D.G. Bariamis, D.K. Iakovidis, D.E. Maroulis, y S.A. Karkanis. An FPGA-based architecture for real time image feature extraction. En *17th International Conference on Pattern Recognition, ICPR'04*, páginas I: 801–804, 2004.

[BK00] V. Bhaskaran y K. Konstantinides. *IMAGE AND VIDEO COMPRESSION: Algorithms and Architectures*. Kluwer Academic Publishers, 2000.

[BL00] A. Bevilacqua y E. Loli. Parallel image restoring on parallel and distributed computers. *Parallel Computing*, 26(4):459–506, 2000.

[Bla01] Nicolas Blanc. CCD versus CMOS: Has CCD imaging come to an end. *Photogrammetric Week*, 2001.

[BM04] S. Baker y I. Matthews. Lucas-kanade 20 years on: A unifying framework. *International Journal on Computer Vision*, 56(3):221–255, February 2004.

[BMRA02] J. Batlle, A. Martí, P. Ridao, y J. Amat. A new FPGA/DSP-based parallel architecture for real.time image processing. *Real-Time Imaging*, 8(5):345–356, October 2002.

[BN07] I. Bouchrika y M.S. Nixon. Model-based feature extraction for gait analysis and recognition. En *Model-based Imaging, Rendering, Image Analysis and Graphical special Effects, MIRAGE'07*, páginas 150–160, 2007.

[BNS+06] L.M. Bergasa, J. J. Nuevo, M.A. Sotelo, R. Barea, y M.E. Lopez. Real-time system for monitoring driver vigilance. *IEEE Transactions on Intelligent Transportation Systems*, 7(1):63–77, March 2006.

[Bol00] José A. Boluda. *Arquitectura de procesamiento de imágenes basada en lógica reconfigurable para navegación de vehículos autónomos con visión foveal*. Tesis Doctoral, Universitat de Valencia, 2000.

[Bou03] N.G. Bourbakis. Bio-imaging and bio-informatics. *IEEE Transactions on Systems, Man and Cybernetics, Part .*, 33(5):726–727, October 2003.

[BR05] M. Barbaro y L. Raffo. A low-power integrated smart sensor with on-chip real-time image processing capabilities. *EURASIP Journal on Applied Signal Processing, JASP*, 2005(7):1062–1070, 2005.

[BW05] A. Bruhn y J. Weickert. Lucas/kanade meets horn/schunck: Combining local and global optic flow methods. *International Journal of Computer Vision*, 61(3):211–231, February 2005.

[Cam94] T. A. Camus. *Real-Time Optical Flow*. Tesis Doctoral, Brown University, 1994.

[Cam97] T.A. Camus. Real-time quantized optical flow. *of Real-Time Imaging (special issue on Real-Time Motion Analysis)*, 3(3):71–86, December 1997.

[CC04] Miguel Correia y Aurelio Campilho. A pipelined real-time optical flow algorithm. En *International Conference, ICIAR 2004*, páginas 372–380, 2004.

[Cha01] J. Chamorro. *Desarrollo de modelos computacionales de representación de secuencias de imágenes y su aplicación a la estimación del movimiento*. Tesis Doctoral, Universidad de Granada, 2001.

[CKC05] D. Chwa, J. Kang, y J.Y. Choi. Online trajectory planning of robot arms for interception of fast maneuvering object under torque and velocity constraints. *IEEE Transactions on Systems, Man and Cybernetics*, 35(6):831–843, November 2005.

[CLK+00] R. Collins, A. Lipton, T. Kanade, H. Fujiyoshi, D. Duggins, Y. Tsin, D. Tolliver, N. Enomoto, O. Hasegawa, P. Birt, y L. Wixson. A system for video surveillance and monitoring. Informe Técnico CMU-RI-TR-00-12, Robotics Institute, Carnegie Mellon University, May. 2000.

[CM01] P. Cobos y F Monasterio. FPGA implementation of camus correlation optical flow algorithm for real time images. En *Proceeding of International Conference on Vision Interface*, páginas 7–9, 2001.

[CM03] P. Cobos y F Monasterio. FPGA implementation of santos-victor optical flow algorithm for real time image processing: an useful attempt. En *The International Society for Optical Engineering, VLSI Circuits and Systems, SPIE*, páginas 23–31, 2003.

[CNBZ05] N. Conci, F.G.B. Natale, J. Bustamante, y S. Zangherati. A wireless multimedia framework for the management of emergency situations in automotive applications: The aider system. *Signal Processing: Image Communication*, 20(9-10):907–926, October 2005.

[Cob01] Pedro Cobos. *Arquitectura Hardware para la Extracción de Invariantes Ópticos, a partir del Flujo Óptico, en Tiempo Real.* Tesis Doctoral, Universidad Politécnica de Madrid, 2001.

[Cor02] Altera Corporation. PCI Megacore Function, Reference Design. Informe Técnico RD-PCIMT32-1.1.0, Altera, September 2002.

[Cor03a] Altera Corporation. SDRAM Reference Design: PCI to DDR. Informe Técnico AN-223, Altera, May 2003.

[Cor03b] Altera Corporation. Stratix PCI Development Board, Data Sheet. Informe Técnico DS-PCIDVBD-2.0, Altera, September 2003.

[Cor04] Altera Corporation. Stratix Device Handbook, Volume 1. Informe Técnico S5V1-3.4, Altera, September 2004.

[Cor05a] Altera Corporation. DDR & DDR2 SDRAM Controller, Compiler User Guide. Informe Técnico UG-DDRSDRAM-3.3.0, Altera, October 2005.

[Cor05b] Altera Corporation. PCI Compiler, User Guide. Informe Técnico UG-PCICOMPILER-4.1.0, Altera, October 2005.

[Cor05c] Altera Corporation. PCI High-Speed Development Kit, Stratix Professional Edition Getting Started User Guide. Informe Técnico D-KIT-1.1.0, Altera, October 2005.

[Cor05d] Altera Corporation. Stratix Device Handbook, Volume 2. Informe Técnico S5V2-3.4, Altera, July 2005.

[Día96] María E. Díaz. *Estimación de Movimiento en secuencias de Imágenes mediante la Detección y Encaje de Puntos Relevantes. Aplicación al seguimiento de Vehículos para el Análisis de Trayectorias*. Tesis Doctoral, Universitat de Valencia, 1996.

[Dal05] Dave L. Dalsa. CMOS vs. CCD: Maturing technologies, maturing markets. *IEEE Trans. Circuits and Systems for Video Technology, CirSysVideo*, 15(7):947–952, July 2005.

[DCSN04] Antonio Dopico, Miguel Correia, Jorge Santos, y Luis Nunes. Distributed computation of optical flow. En *4th International Conference, Computational Science, ICCS 2004*, páginas 380–387, 2004.

[DDB04] P. Dewilde, K. Diepold, y W. Banberger. Optical flow computation and time-varying system theory. En *16th International Symposium on Mathematical Theory of Networks and Systems 04*, páginas I: 20–27, 2004.

[DMC+07] T. Deselaers, H. Muller, P. Clough, H. H. Ney, y T.M. Lehmann. The CLEF 2005 automatic medical image annotation task. *International Journal of Computer Vision, IJCV*, 74(1):51–58, August 2007.

[DRM+06] J. Diaz, E. Ros, S. Mota, F. Pelayo, y E.M. Ortigosa. Subpixel motion computing architecture. *IEE Proceedings-Vision Image and Signal Processing, VISP*, 153(6):869–880, December 2006.

[DRP+06] J. Diaz, E. Ros, F. Pelayo, E.M. Ortigosa, y S. Mota. FPGA-based real-time optical-flow system. *IEEE Transactions on Circuits and Systems for Video Technology*, 16(2):274–279, February 2006.

[DYMS06] A. Duci, A.J. Yezzi, S.K. Mitter, y S. Soatto. Region matching with missing parts. *Image and Vision Computing, IVC*, 24(3):271–277, March 2006.

[FA91] W.T. Freeman y E.H. Adelson. The design and use of steerable filters. *Patter Analysis and Machine Intelligence, IEEE Transactions*, 13(9):891–906, September 1991.

[Fau93] O.D. Faugeras. *Three-Dimensional Computer Vision: A Geometric Viewpoint*. MIT Press, 1993.

[FCD01] M. Fleury, A.F. Clark, y A.C. Downton. Evaluating optical-flow algorithms on a parallel machine. *Image and Vision Computing IVC*, 19(3):131–143, February 2001.

[FII+06] S. Fukui, Y. Iwahori, H. Itoh, H. Kawanaka, y R.J. Woodham. Robust background subtraction for quick illumination changes. En *Advances in Image and Video Technology, PSIVT06*, páginas 1244–1253, 2006.

[FJ90] D. J. Fleet y A. D. Jepson. Computation of component image velocity from local phase information. *International Journal of Computer Vision*, 5(1):77–104, January 1990.

[FJ03] David A. Forsyth y Ponce Jean. *Computer Vision: A Modern Approach*. Prentice Hall, 2003.

[Fly95] M. J. Flynn. *Computer Architecture: Pipelined and Parallel Processor Design*. Jones and Bartlett, 1995.

[FM04] J. Fung y S. Mann. Using multiple graphics cards as a general purpose parallel computer: applications to computer vision. En *17th International Conference on Pattern Recognition, ICPR04*, páginas I: 805–808, 2004.

[FMP05] G.L. Foresti, C. Micheloni, y C. Piciarelli. Detecting moving people in video streams. *International Journal of Intelligent Systems Technologies and Applications*, 26(14):2232–2243, 2005.

[FS05] Dietmar. Fey y Daniel. Schmidt. Marching-pixels: a new organic computing paradigm for smart sensor processor arrays. En *Conference On Computing Frontiers*, páginas 1–9, 2005.

[FSVG07] R. Fransens, C. Strecha, y L.J. Van Gool. Optical flow based super-resolution: A probabilistic approach. *Computer Vision and Image Understanding, CVIU*, 106(1):106–115, April 2007.

[GED04] Jing Guo, Chng Eng, y Rajan Deepu. Foreground motion detection by difference-based spatial temporal entropy image. En *Annual Technical Conference of IEEE region 10, TENCON'04*, páginas 379–382, 2004.

[GG02] Jun Ge y Mirchandani Gagan. A new hybrid block-matching motion estimation algorithm. En *IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, páginas IV–4190, 2002.

[Gha03] Mohammed Ghanbari. *Standard Codecs: Image Compression to Advanced Video Coding*. IEE Telecomunication Series 49, 2003.

[Gib86] James. J. Gibson. *The ecological approach to visual perception*. Lawrence Erlbaum Associates, Publishers, 1986.

[Gle02] Andrew Glennerster. Computational theories of vision. *Current Biology*, 12(20):R682–R685, Octuber 2002.

[GMN+98] B. Galvin, B. McCane, K. K. Novins, D. Mason, y S. Mills. Recovering motion fields: An evaluation of eight optical flow algorithms. En *British Machine Vision Conference*, páginas 195–204, 1998.

[Gon99] Javier Gonzalez. *Visión por Computador*. Thomson Paraninfo, 1999.

[GSS07] U. Grenander, A. Srivastava, y S. Saini. A pattern-theoretic characterization of biological growth. *IEEE Trans. Medical Imaging, MedImg*, 26(5):648–659, May 2007.

[Guh85] Bhaskar Guharoy. Data structure management in a data flow computer system. Informe Técnico MIT/LCS/TR-355, Massachusetts Institute of Technology, May. 1985.

[GVH02] Temujin Gautama y Marc M. Van-Hulle. A phase-based approach to the estimation of optical flow field spatial filtering. *IEEE Transactions on Neural Networks*, 13(5):1127–1136, September 2002.

[Hor86] B.K.P. Horn. *Robot Vision*. MIT Press, 1986.

[How05] N.R. Howe. Flow lookup and biological motion perception. En *International Conference in Image Processing, ICIP05*, páginas III: 1168–1171, 2005.

[HS81] B.K.P. Horn y B.G. Schunck. Determining optical flow. *Artificial Intelligence*, 17(1-3):185–203, August 1981.

[HS07] K.C. Hui y W.C. Siu. Extended analysis of motion-compensated frame difference for block-based motion prediction error. *IEEE Transactions on Image Processing. IP*, 16(5):1232–1245, May 2007.

[HZ04] R. I. Hartley y A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, 2004.

[II03] A. Imiya y K. Iwawaki. Voting method for the detection of subpixel flow field. *Pattern Recognition Letters, PRL*, 24(1-3):197–214, Jan. 2003.

[Jah90] B. Jahne. Motion determination in space-time images. En *IEEE Proceedings of ECCV90*, páginas 161–173, 1990.

[Jah94] B. Jahne. Analytical studies of low-level motion estimators in space-time images using a unified filter concept. En *Conference on Vision and Pattern Recognition*, páginas 229–236, 1994.

[Jai84] R.C. Jain. Difference and accumulative difference pictures in dynamic scene analysis. *Image and Vision Computing, IVC*, 2(2):99–108, May. 1984.

[JN79] R.C. Jain y H. H. Nagel. On the analysis of accumulative difference pictures from image sequences of real world scenes. *IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI*, 1(2):206–214, Mayo 1979.

[KACB05] Richard P. Kleihorst, Anteneh A. Abbo, Vishal Choudhary, y Harry Broers. Scalable ic platform for smart cameras. *EURASIP Journal on Applied Signal Processing, JASP*, 2005(13):2018–2025, 2005.

[Kis02] Laszlo B. Kish. End of moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(3-4):144–149, December 2002.

[KLW05] S. Kim, M.E. Lewis, y C.C. White. Optimal vehicle routing with real-time traffic information. *IEEE Transactions on Intelligent Transportation Systems*, 6(2):178–188, June 2005.

[KMK06] C. Kim, S. Ma, y C.C.J. Kuo. Fast H.264 motion estimation with block-size adaptive referencing (bar). En *International Conference on Image Processing, ICIP'06*, páginas 57–60, 2006.

[KNL+01] H. Kurino, M. M. Nakagawa, K.W. Lee, T.Ñakamura, Y. Yamada, K. Park, y M. M.Koyanagi. Smart vision chip fabricated using three

dimensional integration technology. *Advances in Neural Information Processing Systems.*, 13:MIT Press, 2001.

[KP06] B.G. Kim y D.J. Park. Novel target segmentation and tracking based on fuzzy membership distribution for vision-based target tracking system. *Image and Vision Computing, IVC*, 24(12):1319–1331, December 2006.

[KSF03] H. Kitagawa, J.P. Scheetz, y A.G. Farman. Comparison of complementary metal oxide semiconductor and charge-coupled device intraoral x-ray detectors using subjective image quality. *PubMed, Dentomaxillofac Radiol*, 32(6):408–411, Nov. 2003.

[LdTRF06] P.E. Lopez-de Teruel, A. Ruiz, y L. Fernandez. Geobot: A high level visual perception architecture for autonomous robots. En *IEEE International Conference on Computer Vision Systems*, página 19, 2006.

[LHH+98] H. Liu, T.H. Hong, M. Herman, T.A. Camus, y R. Chellappa. Accuracy vs. efficiency trade-offs in optical flow algorithms. *CVIU*, 72(3):271–286, December 1998.

[LK81] B.D. Lucas y T. Kanade. An iterative image registration technique with an application to stereo vision. En *Proc. DARPA81 Image Understanding Workshop*, páginas 121–130, 1981.

[LLLL07] M.J. Lee, A.S. Lee, D.K. Lee, y S.Y. Lee. Video representation with dynamic features from multi-frame frame-difference images. En *IEEE Workshop on Motion and Video Computing, Motion07*, páginas 28–28, 2007.

[LT05] H. Lanteri y C. Theys. Restoration of astrophysical images: The case of poisson data with additive gaussian noise. *EURASIP Journal on Applied Signal Processing, JASP*, 2005(15):2500–2513, 2005.

[Luc03] M. J. Lucena. *Uso del flujo óptico en algoritmos probabilísticos de seguimiento.* Tesis Doctoral, Universidad de Granada, 2003.

[LWEG07] X. Lu, Y. Wang, E. Erkip, y D. J. Goodman. Total power minimization for multiuser video communications over CDMA networks. *IEEE Transaction on Circuits and Systems for Video Technology, CirSysVideo*, 17(6):674–685, June 2007.

[Ma03] Jing Ma. Signal and image processing via reconfigurable computing. En *Proceedings of the First Workshop on Information and Systems Technology 2003*, páginas 001–006, 2003.

[MCF+06] Giorgio Metta, Laila Craighero, Luciano Fadiga, Kerstin Rosander, Giulio Sandini, David Vernon, y Claes Hofsten. Robotcub development of a cognitive humanoid cub. Informe Técnico European Commission FP6 Project IST-004370, Uppsala University, April 2006.

[MFN06] Giorgio Metta, Paul Fitzpatrick, y Lorenzo Natale. Yarp: Yet another robot platform. *International Journal of Advanced Robotic Systems*, 3(1):043–048, 2006.

[MJE07] Tomasi Matteo, Díaz Javier, y Ros Eduardo. Real time architectures for moving-objects tracking. En *Applied Reconfigurable Computing. ARC'07.*, páginas 365–372, 2007.

[MKN04] M. Meribout, L. Khriji, y M.Ñakanishi. A robust hardware algorithm for real-time object tracking in video sequences. *Real Time Imaging*, 10(3):145–159, June 2004.

[MM04] A. Mitiche y A.R. Mansouri. On convergence of the horn and schunck optical-flow estimation method. *Image Processing*, 13(6):848–852, June 2004.

[MMN06] D.A. Migliore, M. Matteucci, y M.Ñaccari. A revaluation of frame difference in fast and robust motion detection. En *International Workshop on Visual Surveillance and Sensory Networks, VSSN'06*, páginas 215–218, 2006.

[Moi00] Alireza Moini. *VISION CHIPS*. Kluwer Academic Publishers, 2000.

[MRAE03] Selene Maya-Rueda y Miguel Arias-Estrada. Processor for real-time optical flow computation. En *Proceeding of FPL03*, páginas 1103–1106, 2003.

[MRD+06] S. Mota, E. Ros, J. Díaz, R. Agis, y F. de Toro. Bio-inspired motion-based object segmentation. *Lecture Notes in Computer Science, Springer-Verlag*, 4141(8):196–205, September 2006.

[Mye03] P.J. Myerscough. Guiding optical flow estimation. En *British Machine Vision Conference*, páginas 10–16, 2003.

[MZC+05] J.L. Martin, A. Zuloaga, C. Cuadrado, J. Lazaro, y U. Bidarte. Hardware implementation of optical flow constraint equation using FPGA's. *Computer Vision and Image Understanding*, 98(3):462–490, June 2005.

[Nag90] H.H. Nagel. Extending the oriented smoothness constraint into the temporal domain and the estimation of derivatives of optical flow. En *1st European Conference on Computer Vision*, páginas 139–148, 1990.

[OA03] R. Oi y K. Aizawa. Wide dynamic range imaging by sensitivity adjustable cmos image sensor. En *International Conference on Image Processing, ICIP 2003. Proceedings. 2003*, páginas II: 583–586, 2003.

[OD07] M. Oral y U. Deniz. Centre of mass model: A novel approach to background modelling for segmentation of moving objects. *Image and Vision Computing, IVC*, 25(8):1365–1376, August 2007.

[OMTO02] R. Okada, A. Maki, Y. Taniguchi, y K. Onoguchi. Temporally evaluated optical flow: study on accuracy. En *16th International Conference on Pattern Recognition, ICPR02*, páginas I: 343–347, 2002.

[PAE04] Fernando Pardo, Boluda Jose A., y De-Ves Esther. Feature extraction and correlation for time-to-impact segmentation using log-polar images. *Computational Science and Its Applications, ICCSA*, 3046(2-3):887–895, April 2004.

[Paj01] Gonzalo Pajares. *Visión por Computador: Imágenes Digitales y Aplicaciones*. Ra-ma, 2001.

[PGP+00] S.B. Paurazas, J.R. Geist, F.E. Pink, M.M. Hoen, y H.R. Steiman. Comparison of diagnostic accuracy of digital imaging by using ccd and cmos-aps sensors with e-speed film in the detection of periapical bony lesions. *PubMed, Oral Surg Oral Med Oral Pathol Oral Radiol Endod*, 89(3):356–362, Mar. 2000.

[PGPO94] M. Proesmans, L. V. Gool, E. Pauwels, y A. Oosterlinck. Determination of optical flow and its discontinuities using non-linear diffusion. En *3rd European Conference on Computer Vision, ECCV'94*, páginas 295–304, 1994.

[Pic04] M. Piccardi. Background subtraction techniques: a review. *IEEE International Conference on Systems, Man and Cybernetics*, 4(4):3099–3104, Oct. 2004.

[PLV96] E. J. Posnak, R. G. Lavender, y H. M. Vin. Adaptive pipeline: an object structural pattern for adaptive applications. En *Proceedings of the 3rd Pattern Languages of Programming Conference*, páginas 1–11, 1996.

[QZ91] G. Quenot y B. Zavidovique. A data-flow processor for real-time low-level image processing. En *IEEE Custom Integrated Circuits Conference*, páginas 12.4/1–12.4/4, 1991.

[RAAKB05] Richard J. Radke, Srinivas Andra, Omar Al-Kofahi, y Roysam Badrinath. Image change detection algorithm: A systematic survey. *IEEE Transaction on Image Procesing*, 14(3):294–307, March 2005.

[RB07] S. Roth y M.J. Black. On the spatial statistics of optical flow. *International Journal of Computer Vision*, 74(1):33–50, August 2007.

[RJK06] T. Ryu, G. Jung, y J.N. Kim. A fast full search algorithm for motion estimation using priority of matching scan. En *International Conference on Image Analysis and Recognition, ICIAR'06*, páginas I:571–579, 2006.

[RV95] Reinhard Rauscher y S. Venzke. An ASIC performing image processing tasks in realtime. En *EUROMICRO'95*, páginas 000–008, 1995.

[SD04] Alan Stocker y Rodney Douglas. Analog integrated 2-D optical flow sensor with programmable pixels. En *IEEE International Symposium on Circuits And System, ISCAS'04*, páginas 9–12, 2004.

[SFK97] D. Sima, T. Fountain, y P. Karsuk. *Advanced Computer Architectures: A Design Space Approach*. Addison-Wesley Professional, 1997.

[SGFBP06a] Julio C. Sosa, Rocío Gómez-Fabela, José A. Boluda, y Fernando Pardo. Change-driven image processing architecture with adaptive threshold for optical-flow computation. En *3rd International Conference on ReConFigurable Computing and FPGA's, ReConFig'06*, páginas 237–244, 2006.

[SGFBP06b] Julio C. Sosa, Rocio Gomez-Fabela, Jose A. Boluda, y Fernando Pardo. FPGA implementation of change-driven image processing architecture for optical flow computation. En *16 Th International Conference On Field Programmable Logic and Applications, FPL'06*, páginas 663–666, 2006.

[SGPB04] Julio C. Sosa, R. Gómez, F. Pardo, y J.A. Boluda. Visión log-polar basada en una FPGA y el bus PCI para aplicaciones en tiempo-real. En *Congreso Internacional de Cómputo Reconfigurable y FPGA's, ReConFig'04*, páginas 210–219, 2004.

[SHS04] P. Sangi, J. Heikkila, y O. Silven. Motion analysis using frame differences with spatial gradient measures. En *International Conference on Pattern Recognition, ICPR'04*, páginas IV: 733–736, 2004.

[Sim93] E. Simoncelli. *Distributed Representation and Analysis of Visual Motion*. Tesis Doctoral, MIT, 1993.

[Sin90] A. Singh. *Optic Flow Computation: A Unified Perspective*. IEEE Press, 1990.

[SK02] D. Stichling y B. Kleinjohann. CV-SDF - a model for real-time computer vision applications. En *6th IEEE Workshop on Applications of Computer Vision, WACV 2002*, páginas 325–329, 2002.

[SKG01] Frank Seinstra, Dennis Koelma, y Jan-Mark Geusebroek. A software architecture for user transparent parallel image processing on MIMD computers. *Lecture Notes in Computer Science*, 2150:653–661, 2001.

[SKG02] Frank Seinstra, Dennis Koelma, y Jan-Mark Geusebroek. A software architecture for user transparent parallel image processing. *Parallel computing in image and video processing*, 28(7-8):967–993, August 2002.

[SKMD03] Karthikeyan Sankaralingam, Stephen W. Keckler, William R. Mark, y Burger Doug. Universal mechanisms for data-parallel architectures. En *36th Annual International Symposium on Microarquitecture*, páginas 303–315, 2003.

[SKS01] P.R. Schrater, D.C. Knill, y E.P. Simoncelli. Perceiving visual expansion without optic flow. *Nature*, 1(410):816–819, April 2001.

[SS04] L. Spencer y M. Shah. Water video analysis. En *International Conference on Image Processing*, páginas IV: 2705–2708, 2004.

[SSCW98] P.A. Smith, D. Sinclair, R. Cipolla, y K. Wood. Effective corner matching. En *9th British Machine Vision Conference, BMVC98*, páginas 545–556, 1998.

[Sta00] Williams Stallings. *Organización y Arquitectura de Computadores*. Prentice Hall, 2000.

[Sto06] Alan Stocker. Analog integrated 2-D optical flow sensor. *Analog Integrated Circuits and Signal Processing, Springer*, 46(2):121–138, January 2006.

[SV05] A. Sanfelui y J. J. Villanueva. An approach of visual motion analysis. *Pattern Recognition Letters*, 26(3):355–368, February 2005.

[Tag07] M. Tagliasacchi. A genetic algorithm for optical flow estimation. *Image and Vision Computing, IVC*, 25(2):141–147, February 2007.

[TB81] W.B. Thompson y S.T. Barnard. Lower-level estimation and interpretation of visual motion. *International Journal of Computer Vision*, 14(8):20–28, June 1981.

[TB01] Russell Tessier y Wayne Burleson. Reconfigurable computing for digital signal processing: A survey. *Journal of VLSI Signal Processing*, 28(1-2):7–27, May 2001.

[TcAA04] B. Ugur Töreyin, A. Enis Çetin, Anil Aksay, y M. Bilgay Akhan. Moving region detection in compressed video. En *ISCIS*, páginas 381–390, 2004.

[TCP04] B. Tedla, S.D. Cabrera, y N.J. Parks. Analysis and restoration of desert/urban scenes degraded by the atmosphere. En *6th IEEE Southwest Symposium on Image Analysis and Interpretation*, páginas 11–15, 2004.

[THAE04] C. Torres-Huitzil y M. Arias-Estrada. Real-time image processing with a compact FPGA-based systolic architecture. *Real Time Imaging*, 10(3):177–187, June 2004.

[Til99] J.C. Tilton. A recursive pvm implementation of an image segmentation algorithm performance results comparing the hive and cray t3e. En *Frontiers of Massively Parallel Computation*, páginas 146–153, 1999.

[TJN98] A. L. Tabatabai, R. S. Jasindchi, y T. T. Naveen. Motion estimation methods for video compression – A review. *Elsevier Science Ltd, J. Franklin Inst.*, 335B(8):1411–1441, Jan. 1998.

[TLCH05] C.H. Teng, S.H. Lai, Y.S. Chen, y W.H. Hsu. Accurate optical flow computation under non-uniform brightness variations. *Computer Vision and Image Understanding, CVIU*, 97(3):315–346, March 2005.

[TM02] B.J. Tordoff y D.W. Murray. Guided sampling and consensus for motion estimation. En *European Conference on Computer Vision, ECCV02*, página I: 82 ff., 2002.

[Váz96] Fernando Vázquez. *Segmentación de Imágenes en grafos de contorno. Aplicación a la estimación de la profundidad y el movimiento relativo para un robot móvil autónomo.* Tesis Doctoral, Universidad de Vigo, 1996.

[Wan01] Ching-Chun Wang. *A Study of CMOS Technologies for Image Sensor Applications.* Tesis Doctoral, Massachusetts Institute of Technology, 2001.

[WB04] J. Wills y S. Belongie. A feature-based approach for determining dense long range correspondences. En *European Conference on Computer Vision, ECCV04*, páginas Vol III: 170–182, 2004.

[WMVB94] A. B. Wiegand Metha, A. J. Vingrys, y D. R. Badcock. Detection and discrimination of moving stimuli: the effects of color, luminance and eccentricity. *Optical Society of America. A, Optics, image science, and vision*, 11(11):1697–1709, Jun. 1994.

[WSG05] T. Wiegand, E.G. Steinbach, y B. Girod. Affine multipicture motion-compensated prediction. *IEEE Transactions on Circuits and Systems for Video Technology*, 15(2):197–209, February 2005.

[XCS+06] J. Xiao, H. Cheng, H.S. Sawhney, C. Rao, y M. Isnardi. Bilateral filtering-based optical flow estimation with occlusion detection. En

European Conference on Computer Vision, ECCV06, páginas I: 211–224, 2006.

[YA81] S. Yalamanchili y J.K. Aggarwal. Motion and image differencing. En *Pattern Recognition and Information Processing, PRIP81*, páginas 211–216, 1981.

[YB03] Albert Yeung y Nick Barnes. Active monocular fixation using log-polar sensor. En *Proc. Australian Conference on Robotics and Automation*, páginas II: 583–586, 2003.

[YB05] Albert Yeung y Nick Barnes. Efficient active monocular fixation using the log-polar sensor. *International Journal of Intelligent Systems Technologies and Applications*, 1(1-2):157–173, 2005.

[YCH07] H. Yalcin, R. Collins, y M. Hebert. Background estimation under rapid gain change in thermal imagery. *Computer Vision and Image Understanding, CVIU*, 106(2-3):148–161, May 2007.

[Yea02] Mohammed Yeasin. Optic flow in log-mapped image plane: A new approach. *IEEE Transaction on Pattern Analysis and Machine Intelligentce*, 24(1):125–131, January 2002.

[YZKJ05] Cheng Yun, Wang Zhiying, Dai Kui, y Guo J. A fast motion estimation algorithm based on diamond and triangle search patterns. *Lecture Notes in Computer Science*, 3522:419–426, May 2005.

[ZAS06] J. Zan, M.O. Ahmad, y M.N.S. Swamy. Comparison of wavelets for multiresolution motion estimation. *IEEE Transaction on Circuits and Systems for Video Technology, CirSysVideo*, 16(3):439–446, March 2006.

Publicaciones relacionadas con el presente trabajo

[PBS03a] F. Pardo, J. A. Boluda y J. C. Sosa. Transformación Log-Polar en FPGAs Utilizando CORDIC. En *III Jornadas sobre Computación Reconfigurable y Aplicaciones, JCRA'03*, páginas: 99-106, Madrid, España, Septiembre 2003.

[PBS03b] F. Pardo, J. A. Boluda y J. C. Sosa. A log-polar image processing system on a chip. En *XVIII Design of Circuits and Integrated Systems, DCIS03*, páginas 449-454, Ciudad Real, España, November 2003.

[SPBG04] J. C. Sosa, F. Pardo, J. A. Boluda y R. Gómez. Desarrollo de una interfaz PCI para un sistema de visión en una FPGA. En *IV Jornadas sobre Computación Reconfigurable y Aplicaciones, JCRA'04*, páginas 531-540, Barcelona, España, Septiembre 2004.

[SGPB04] J. C. Sosa, R. Gómez, F. Pardo y J. A. Boluda. Visión Log-Polar Basada en una FPGA y el Bus PCI para aplicaciones en Tiempo-Real. En *Congreso Internacional de Cómputo Reconfigurable y FPGAs, RECONFIG'04*, páginas 210-219, Colima, México, Septiembre 2004.

[GSP05] R. Gómez, J. C. Sosa, y F. Pardo. Detección de movimiento en imágenes comprimidas utilizando la transformada Wavelet. En *V Jornadas de Computación Reconfigurable y Aplicaciones. JCRA'2005*, páginas 139-144, Granada, España, Septiembre 2005.

[PBBDS05] Fernando Pardo, Jose A. Boluda, Xaro Benavent, Juan Domingo y Julio C. Sosa. Circle detection and tracking speed-up based on change-driven image processing. En *ICGST International Conference on Graphics, Vision and Image Processing, GVIP'05*, páginas 131-136, Cairo, Egipto, Diciembre 2005.

[SGFBP06] Julio C. Sosa, Rocío Gomez-Fabela, José A. Boluda y Fernando Pardo. FPGA Implementation of Change-Driven Image Processing Architecture for Optical Flow Computation. En *16 Th International Conference On Field Programmable Logic and Applications, FPL'06*, páginas 663-666, Madrid, España, Agosto 2006.

[SGFBP06a] Julio C. Sosa, Rocio Gómez-Fabela, Jose A. Boluda y Fernando Pardo. Change-Driven Image Processing Architecture with Adaptive Threshold for Optical-Flow Computation. En *The 3rd International Conference on ReConFigurable Computing and FPGA's, RECONFIG'06*, páginas 237-244, San Luís Potosí, México, Septiembre 2006.

Parte VI

Apéndices

Apéndice A

Código VHDL

El código VHDL que se muestra en este apéndice corresponde a la versión final de los diferentes módulos que conforman la arquitectura para el procesado del flujo óptico guiado por cambios, descrita en el presente trabajo de investigación. Se ha realizado y modificado más código que el aquí presentado, entre los cuales está el utilizado para la simulación con el banco de pruebas. Debido a la gran cantidad de líneas de código utilizado es imposible desplegarlo en este apartado. Por tal razón se incluye un CD que contiene tanto las fuentes del programa *testbench* como el del programa de la interfaz realizado en C++ Builder.

Los programas VHDL que se muestran en las siguientes secciones constituyen básicamente las etapas independientes de cada módulo y el conjunto de máquina de estados utilizadas para llevar a cabo la escritura y lectura de la DDR SDRAM. También se expone el módulo de control general, lo que supone la exposición del sistema para el procesado de las imágenes diseñado en este trabajo.

A.1. Código VHDL del módulo `LUT/Grad`

El módulo de detección de cambios y el del gradiente fueron fusionados en un único módulo denominado `LUT/Grad`. Este módulo esta constituido por un conjunto de sumas y restas. Ingresan 8 píxeles simultáneamente y generan cuatro componentes de salida, los gradientes I_x, I_y, I_t, de 9 bits, y un píxel de cambios ch.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY gradiente IS
PORT(
clk,reset : IN STD_LOGIC;
a,b,c,d,e,f,g,h : IN STD_LOGIC_VECTOR( 7 DOWNTO 0);
Ch   : OUT STD_LOGIC;
Ix,Iy,It : OUT STD_LOGIC_VECTOR( 8 DOWNTO 0));

END gradiente;

ARCHITECTURE a OF gradiente IS

signal Eax, Eay, Eat : STD_LOGIC_VECTOR(10 DOWNTO 0);
signal dif, mag : STD_LOGIC_VECTOR( 8 DOWNTO 0);
constant umbral : STD_LOGIC_VECTOR( 2 DOWNTO 0):="001"; -- 7 =<(Entero)> 0 ;
signal tmp : STD_LOGIC;

BEGIN
Eax<=('0'&'0'&'0'& a) - ('0'&'0'&'0'& b)
+('0'&'0'&'0'& c) - ('0'&'0'&'0'& d)
+('0'&'0'&'0'& e) - ('0'&'0'&'0'& f)
+('0'&'0'&'0'& g) - ('0'&'0'&'0'& h);

Eay<=('0'&'0'&'0'& d) - ('0'&'0'&'0'& b)
+('0'&'0'&'0'& c) - ('0'&'0'&'0'& a)
+('0'&'0'&'0'& h) - ('0'&'0'&'0'& f)
+('0'&'0'&'0'& g) - ('0'&'0'&'0'& e);

Eat<=('0'&'0'&'0'& f) - ('0'&'0'&'0'& b)
+('0'&'0'&'0'& h) - ('0'&'0'&'0'& d)
+('0'&'0'&'0'& e) - ('0'&'0'&'0'& a)
+('0'&'0'&'0'& g) - ('0'&'0'&'0'& c);

--Para general el bit de la tabla de cambios
Dif <=  ('0'& e) - ('0'& a);
mag<=Dif WHEN Dif(8)='0' ELSE NOT Dif +"000000001";
tmp<='1' WHEN (mag > umbral) ELSE '0';

PROCESS (clk,reset)

BEGIN
IF reset = '0' THEN
Ix<="000000000";
Iy<="000000000";
It<="000000000";
Ch<='0';
ELSIF (clk 'EVENT AND clk='1') THEN
Ix<=Eax(10 DOWNTO 2);
Iy<=Eay(10 DOWNTO 2);
It<=Eat(10 DOWNTO 2);
```

```
Ch<=tmp;
END IF;

END PROCESS;
END a;
```

El siguiente código realiza la misma función que el anterior pero en vez de procesar una sola componente para cada gradiente y un bit de detección de cambios, ahora procesa 32 píxeles simultáneamente. Por lo tanto, la función proporcionará 32 componentes de salida, 8 componentes para cada gradiente (I_x, I_y, I_t), de 9 bits, y ocho píxeles de cambios *ch*. El código que describe esta función es:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;


ENTITY grad8 IS
PORT(
clk,reset : IN STD_LOGIC;
q11, q12, q21, q22 : IN STD_LOGIC_VECTOR( 63 DOWNTO 0);
Ch0,Ch1,Ch2,Ch3,Ch4,Ch5,Ch6,Ch7 : OUT STD_LOGIC;
Ix1,Iy1,It1 : OUT STD_LOGIC_VECTOR( 8 DOWNTO 0);
Ix2,Iy2,It2 : OUT STD_LOGIC_VECTOR( 8 DOWNTO 0);
Ix3,Iy3,It3 : OUT STD_LOGIC_VECTOR( 8 DOWNTO 0);
Ix4,Iy4,It4 : OUT STD_LOGIC_VECTOR( 8 DOWNTO 0);
Ix5,Iy5,It5 : OUT STD_LOGIC_VECTOR( 8 DOWNTO 0);
Ix6,Iy6,It6 : OUT STD_LOGIC_VECTOR( 8 DOWNTO 0);
Ix7,Iy7,It7 : OUT STD_LOGIC_VECTOR( 8 DOWNTO 0);
Ix0,Iy0,It0 : OUT STD_LOGIC_VECTOR( 8 DOWNTO 0));

END grad8;


ARCHITECTURE a OF grad8 IS

COMPONENT gradiente
PORT(
clk,reset : IN STD_LOGIC;
a,b,c,d,e,f,g,h : IN STD_LOGIC_VECTOR( 7 DOWNTO 0);
Ch   : OUT STD_LOGIC;
Ix,Iy,It : OUT STD_LOGIC_VECTOR( 8 DOWNTO 0));
END COMPONENT;

----Pixeles temporales, para utilizarlos entre una palabra y la siguiente.
signal q12_tmp, q11_tmp : STD_LOGIC_VECTOR(7 downto 0);
signal q22_tmp, q21_tmp : STD_LOGIC_VECTOR(7 downto 0);

BEGIN

PROCESS (clk,reset)

BEGIN
IF reset = '0' THEN
q12_tmp<="00000000";
q11_tmp<="00000000";
q22_tmp<="00000000";
```

```
q21_tmp<="00000000";

ELSIF (clk 'EVENT AND clk='1') THEN
q12_tmp<=q12(7 downto 0);
q11_tmp<=q11(7 downto 0);
q22_tmp<=q22(7 downto 0);
q21_tmp<=q21(7 downto 0);

END IF;

END PROCESS;

gradiente0 : gradiente PORT MAP(
clk=>clk,
reset=>reset,
a=>q12(63 downto 56),
b=>q12_tmp,
c=>q11(63 downto 56),
d=>q11_tmp,
e=>q22(63 downto 56),
f=>q22_tmp,
g=>q21(63 downto 56),
h=>q21_tmp,
Ch=>Ch0,
Ix=>Ix0,
Iy=>Iy0,
It=>It0
);

gradiente1 : gradiente PORT MAP(..);

gradiente2 : gradiente PORT MAP(..);

gradiente3 : gradiente PORT MAP(..);

gradiente4 : gradiente PORT MAP(..);

gradiente5 : gradiente PORT MAP(..);

gradiente6 : gradiente PORT MAP(..);

gradiente7 : gradiente PORT MAP(..);

END a;
```

A.2. Código VHDL del módulo `Velocidad`

El módulo `Velocidad` esta constituido por un conjunto de módulos que efectúan productos, sumas y divisiones. Aquí sólo se procesa un conjunto componentes para obtener las componentes que conforman al vector del flujo óptico, las componentes *(U, V)*. También se muestran un grupo de registros que sirven para mantener los valores de unas variables y utilizarlos hasta la salida. La razón es que la arquitectura está segmentada para la implementación del circuito divisor utilizado en este módulo.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY velocidad IS
PORT(
clk,reset : IN STD_LOGIC;
vels_start : IN STD_LOGIC;
Ix,Iy,It,LU,LV : IN STD_LOGIC_VECTOR( 8 DOWNTO 0);
vels_ack : OUT STD_LOGIC;
U,V : OUT STD_LOGIC_VECTOR( 8 DOWNTO 0));

END velocidad;


ARCHITECTURE a OF velocidad IS

component termino_D
PORT
(
dataa_0 : IN STD_LOGIC_VECTOR (7 DOWNTO 0) :=  (OTHERS => '0');
dataa_1 : IN STD_LOGIC_VECTOR (7 DOWNTO 0) :=  (OTHERS => '0');
datab_0 : IN STD_LOGIC_VECTOR (7 DOWNTO 0) :=  (OTHERS => '0');
datab_1 : IN STD_LOGIC_VECTOR (7 DOWNTO 0) :=  (OTHERS => '0');
result : OUT STD_LOGIC_VECTOR (16 DOWNTO 0)
);
end component;

component termino_Paux
PORT
(
dataa_0 : IN STD_LOGIC_VECTOR (8 DOWNTO 0) :=  (OTHERS => '0');
dataa_1 : IN STD_LOGIC_VECTOR (8 DOWNTO 0) :=  (OTHERS => '0');
datab_0 : IN STD_LOGIC_VECTOR (8 DOWNTO 0) :=  (OTHERS => '0');
datab_1 : IN STD_LOGIC_VECTOR (8 DOWNTO 0) :=  (OTHERS => '0');
result : OUT STD_LOGIC_VECTOR (17 DOWNTO 0)
);
end component;

component ter_Px
PORT
(
dataa : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
datab : IN STD_LOGIC_VECTOR (17 DOWNTO 0);
result : OUT STD_LOGIC_VECTOR (26 DOWNTO 0)
);
end component;

component ter_Py
```

```
PORT
(
dataa : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
datab : IN STD_LOGIC_VECTOR (17 DOWNTO 0);
result : OUT STD_LOGIC_VECTOR (26 DOWNTO 0)
);
end component;

component div_Rx
PORT
(
clock : IN STD_LOGIC ;
denom : IN STD_LOGIC_VECTOR (16 DOWNTO 0);
numer : IN STD_LOGIC_VECTOR (25 DOWNTO 0);
quotient: OUT STD_LOGIC_VECTOR (25 DOWNTO 0);
remain : OUT STD_LOGIC_VECTOR (16 DOWNTO 0)
);
end component;

component div_Ry
PORT
(
clock : IN STD_LOGIC ;
denom : IN STD_LOGIC_VECTOR (16 DOWNTO 0);
numer : IN STD_LOGIC_VECTOR (25 DOWNTO 0);
quotient: OUT STD_LOGIC_VECTOR (25 DOWNTO 0);
remain : OUT STD_LOGIC_VECTOR (16 DOWNTO 0)
);
end component;



--Señales para el termino D
signal Ex_d, Ey_d : STD_LOGIC_VECTOR (7 DOWNTO 0);
signal Daux, TD, TD1, TD2 : STD_LOGIC_VECTOR(16 DOWNTO 0);

--Señales para el termino P
signal Et_p, Paux, TP, TP1 : STD_LOGIC_VECTOR(17 DOWNTO 0);
signal tmp_Ex,tmp_Ey : STD_LOGIC_VECTOR (8 DOWNTO 0);
signal Px_aux, Py_aux : STD_LOGIC_VECTOR(26 DOWNTO 0);
signal Px, Py : STD_LOGIC_VECTOR(26 DOWNTO 0);
signal PX1, PY1 : STD_LOGIC_VECTOR(25 DOWNTO 0);

signal Rx_aux, Ry_aux : STD_LOGIC_VECTOR(25 DOWNTO 0);
signal Rx,Ry : STD_LOGIC_VECTOR( 8 DOWNTO 0);

--Residuos no usados
signal resx, resy : STD_LOGIC_VECTOR(16 DOWNTO 0);
--Registros para LU y LV
signal LU1,LU2,LU3,LU4,LU5,LU6,LU7,LU8 : STD_LOGIC_VECTOR( 8 DOWNTO 0);
signal LV1,LV2,LV3,LV4,LV5,LV6,LV7,LV8 : STD_LOGIC_VECTOR( 8 DOWNTO 0);
signal ack1,ack2,ack3,ack4,ack5,ack6,ack7,ack8 : STD_LOGIC;

BEGIN

Ex_d<= Ix(7 DOWNTO 0) WHEN Ix(8)='0' ELSE -Ix(7 DOWNTO 0);
Ey_d<= Iy(7 DOWNTO 0) WHEN Iy(8)='0' ELSE -Iy(7 DOWNTO 0);
TD<="00000000000000001" WHEN Daux="00000000000000000" ELSE Daux;

Et_p<= "000000000"&It WHEN It(8)='0' ELSE "111111111"&It;
TP<=Paux + Et_p;

PX1<=(Px(26) & Px(24 DOWNTO 0));
PY1<=(Py(26) & Py(24 DOWNTO 0));
```

```
Rx<=(Rx_aux(25) & Rx_aux(7 DOWNTO 0));
Ry<=(Ry_aux(25) & Ry_aux(7 DOWNTO 0));

PROCESS (clk,reset)
BEGIN
IF reset = '0' THEN
     Px       <= "00000000000000000000000000";
     Py       <= "00000000000000000000000000";
     U        <= "000000000";
     V        <= "000000000";
     ack1     <= '0';
     ack2     <= '0';
     ack3     <= '0';
     ack4     <= '0';
     ack5     <= '0';
     ack6     <= '0';
     ack7     <= '0';
     ack8     <= '0';
     vels_ack <= '0';
     LU1      <= "000000000";
     LU2      <= "000000000";
     LU3      <= "000000000";
     LU4      <= "000000000";
     LU5      <= "000000000";
     LU6      <= "000000000";
     LU7      <= "000000000";
     LV1      <= "000000000";
     LV2      <= "000000000";
     LV3      <= "000000000";
     LV4      <= "000000000";
     LV5      <= "000000000";
     LV6      <= "000000000";
     LV7      <= "000000000";
     LV8      <= "000000000";
     tmp_Ex    <= "000000000";
     tmp_Ey    <= "000000000";
ELSIF (clk 'EVENT AND clk='1') THEN
 ack1 <= vels_start;
     ack2 <= ack1;
     ack3 <= ack2;
     ack4 <= ack3;
     ack5 <= ack4;
     ack6 <= ack5;
     ack7 <= ack6;
     ack8 <= ack7;
     vels_ack <= ack8;
     TD1<=TD;
     TD2<=TD1;
     TP1<=TP;
     tmp_Ex<=Ix;
     tmp_Ey<=Iy;
     Px<=Px_aux;
     Py<=Py_aux;
     --Registros de LU y LV:
     LU1<=LU;
     LU2<=LU1;
     LU3<=LU2;
     LU4<=LU3;
     LU5<=LU4;
     LU6<=LU5;
     LU7<=LU6;
     LU8<=LU7;
     LV1<=LV;
     LV2<=LV1;
```

```
     LV3<=LV2;
     LV4<=LV3;
     LV5<=LV4;
     LV6<=LV5;
     LV7<=LV6;
     LV8<=LV7;
     ---Obtencion de Velocidades
     U<=LU8-Rx;
     V<=LV8-Ry;
   END IF;
END PROCESS;

termino_D_inst : termino_D PORT MAP (
dataa_0  => Ex_d,
dataa_1  => Ey_d,
datab_0  => Ex_d,
datab_1  => Ey_d,
result  => Daux
);

termino_Paux_inst : termino_Paux PORT MAP (
dataa_0  => Ix,
dataa_1  => Iy,
datab_0  => LU,
datab_1  => LV,
result  => Paux
);

ter_Px_inst : ter_Px PORT MAP (
dataa  => tmp_Ex,
datab  => TP1,
result  => Px_aux
);

ter_Py_inst : ter_Py PORT MAP (
dataa  => tmp_Ey,
datab  => TP1,
result  => Py_aux
);

div_Rx_inst : div_Rx PORT MAP (
clock  => clk,
denom  => TD2,
numer  => PX1,
quotient => Rx_aux,
remain  => resx
);

div_Ry_inst : div_Ry PORT MAP (
clock  => clk,
denom  => TD2,
numer  => PY1,
quotient => Ry_aux,
remain  => resy
);

END a;
```

En la mayoría de los casos, como en éste y los programas posteriores, no se presentan todos los **COMPONENT** utilizados en los programas. Sin embargo en el CD anexo se incorporan dichos archivos.

A.3. Código VHDL del módulo `Laplaciana`

El módulo `Laplaciana` realiza la última restricción para el cálculo del flujo óptico. La restricción consiste en un promediado de las velocidades capturadas. En este caso serían las componentes de la velocidad (U, V). El resultado es la componente promedio de sus vecinos. De esta manera se tienen que sumar 8 píxeles, los vecinos de cada componente a evaluar, y efectuar una división. El siguiente código realiza dicho proceso:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY laplaciana IS
  PORT(
       clk, reset : IN  STD_LOGIC;
       a,b,c,d,e,f,g,h : IN  STD_LOGIC_VECTOR( 8 DOWNTO 0);
       Lapla           : OUT STD_LOGIC_VECTOR( 8 DOWNTO 0);
       lapla_start     : IN  STD_LOGIC;
       lapla_ack       : OUT STD_LOGIC);
END laplaciana;

ARCHITECTURE c OF laplaciana IS
 COMPONENT div_por3
PORT
(
denom : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
numer : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
quotient : OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
remain : OUT STD_LOGIC_VECTOR (1 DOWNTO 0)
);
 END COMPONENT;

  SIGNAL a1, b1, c1, d1   : STD_LOGIC_VECTOR(10 DOWNTO 0);
  SIGNAL e1, f1, g1, h1   : STD_LOGIC_VECTOR(10 DOWNTO 0);
  SIGNAL L1, L2   : STD_LOGIC_VECTOR(10 DOWNTO 0);
  SIGNAL basura1, basura2 : STD_logic_VECTOR( 1 DOWNTO 0);
  SIGNAL Lt1, Lt2   : STD_LOGIC_VECTOR(10 DOWNTO 0);
  SIGNAL Laux    : STD_LOGIC_VECTOR( 8 DOWNTO 0);

  BEGIN
    a1<="00"&a WHEN a(8)='0' ELSE "11"&a;
    b1<="00"&b WHEN b(8)='0' ELSE "11"&b;
    c1<="00"&c WHEN c(8)='0' ELSE "11"&c;
    d1<="00"&d WHEN d(8)='0' ELSE "11"&d;
    e1<="00"&e WHEN e(8)='0' ELSE "11"&e;
    f1<="00"&f WHEN f(8)='0' ELSE "11"&f;
    g1<="00"&g WHEN g(8)='0' ELSE "11"&g;
    h1<="00"&h WHEN h(8)='0' ELSE "11"&h;
    L1<= a1 + b1 + c1 + d1;
    L2<= e1 + f1 + g1 + h1;
    Laux<= (Lt1(10)& Lt1(8 DOWNTO 1)) + (Lt2(10 DOWNTO 2));

    PROCESS (clk,reset)
    BEGIN
      IF reset = '0' THEN
        Lapla     <="000000000";
        lapla_ack <= '0';
```

```
      ELSIF (clk 'EVENT AND clk='1') THEN
        Lapla     <= Laux;
        lapla_ack <= lapla_start;
    END IF;
    END PROCESS;

    div_por3_inst : div_por3 PORT MAP (
      numer    => L1,
      denom    => "11",
      quotient => Lt1,
      remain   => basura1
    );

   div_por3_inst_2 : div_por3 PORT MAP (
      numer    => L2,
      denom    => "11",
      quotient => Lt2,
      remain   => basura2
    );
END c;
```

El resultado de salida es sólo una componente. Si se desea realizar un procesado en paralelo que obtenga simultáneamente 8 componentes de salida, es necesario realizar otro programa que realice dicha función. De esta manera el código resultante para este propósito es:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY laplaciana8 IS
  PORT(
       clk, reset        : IN  STD_LOGIC;
       q_11, q_12, q_13      : IN  STD_LOGIC_VECTOR(71 DOWNTO 0);
       L0,L1,L2,L3,L4,L5,L6,L7 : OUT STD_LOGIC_VECTOR(8  DOWNTO 0);
       lapla_start             : IN  STD_LOGIC;
       lapla_ack               : OUT STD_LOGIC);
END laplaciana8;

ARCHITECTURE c OF laplaciana8 IS

  COMPONENT laplaciana
    PORT(
         clk, reset   : IN STD_LOGIC;
         a,b,c,d,e,f,g,h   : IN  STD_LOGIC_VECTOR(8 DOWNTO 0);
         Lapla           : OUT STD_LOGIC_VECTOR(8 DOWNTO 0));
  END COMPONENT;


  SIGNAL q_11a_tmp,q_11b_tmp         : STD_LOGIC_VECTOR(8 DOWNTO 0);
  SIGNAL q_12a_tmp,q_12b_tmp        : STD_LOGIC_VECTOR(8 DOWNTO 0);
  SIGNAL q_13a_tmp,q_13b_tmp        : STD_LOGIC_VECTOR(8 DOWNTO 0);
  SIGNAL L0_aux,L1_aux,L2_aux,L3_aux : STD_LOGIC_VECTOR(8 DOWNTO 0);
  SIGNAL L4_aux,L5_aux,L6_aux,L7_aux : STD_LOGIC_VECTOR(8 DOWNTO 0);
  SIGNAL lapla_tmp : STD_LOGIC;

  BEGIN

   PROCESS (clk,reset)
```

```
  BEGIN
    IF reset = '0' THEN
    L0<="000000000";
    L1<="000000000";
    L2<="000000000";
    L3<="000000000";
    L4<="000000000";
    L5<="000000000";
    L6<="000000000";
    L7<="000000000";
    lapla_ack <= '0';
  ELSIF (clk 'EVENT AND clk='1') THEN
    q_11a_tmp<=q_11(17 DOWNTO  9);
    q_12a_tmp<=q_12(17 DOWNTO  9);
    q_13a_tmp<=q_13(17 DOWNTO  9);
    q_11b_tmp<=q_11( 8 DOWNTO  0);
    q_12b_tmp<=q_12( 8 DOWNTO  0);
    q_13b_tmp<=q_13( 8 DOWNTO  0);
    L0<=L0_aux;
    L1<=L1_aux;
    L2<=L2_aux;
    L3<=L3_aux;
    L4<=L4_aux;
    L5<=L5_aux;
    L6<=L6_aux;
    L7<=L7_aux;
    lapla_tmp <= lapla_start;
    lapla_ack <= lapla_tmp;
  END IF;
  END PROCESS;

 laplaciana0: laplaciana PORT MAP(
    clk => clk,
    reset => reset,
    a   => q_11b_tmp,
    b   => q_12(71 DOWNTO 63),
    c   => q_13b_tmp,
    d   => q_12a_tmp,
    e   => q_11a_tmp,
    f   => q_11(71 DOWNTO 63),
    g   => q_13(71 DOWNTO 63),
    h   => q_13a_tmp,
    Lapla => L0_aux
 );

laplaciana1: laplaciana PORT MAP (
    clk => clk,
    reset => reset,
    a   => q_11(71 DOWNTO 63),
    b   => q_12(62 DOWNTO 54),
    c   => q_13(71 DOWNTO 63),
    d   => q_12b_tmp,
    e   => q_11b_tmp,
    f   => q_11(62 DOWNTO 54),
    g   => q_13(62 DOWNTO 54),
    h   => q_13b_tmp,
    Lapla => L1_aux
 );

laplaciana2: laplaciana PORT MAP (
    clk => clk,
    reset => reset,
    a   => q_11(62 DOWNTO 54),
    b   => q_12(53 DOWNTO 45),
```

```
    c   => q_13(62 DOWNTO 54),
    d   => q_12(71 DOWNTO 63),
    e   => q_11(71 DOWNTO 63),
    f   => q_11(53 DOWNTO 45),
    g   => q_13(53 DOWNTO 45),
    h   => q_13(71 DOWNTO 63),
    Lapla => L2_aux
 );


laplaciana3: laplaciana PORT MAP (
    clk => clk,
    reset => reset,
    a   => q_11(53 DOWNTO 45),
    b   => q_12(44 DOWNTO 36),
    c   => q_13(53 DOWNTO 45),
    d   => q_12(62 DOWNTO 54),
    e   => q_11(62 DOWNTO 54),
    f   => q_11(44 DOWNTO 36),
    g   => q_13(44 DOWNTO 36),
    h   => q_13(62 DOWNTO 54),
    Lapla => L3_aux
 );

laplaciana4: laplaciana PORT MAP (
    clk => clk,
    reset => reset,
    a   => q_11(44 DOWNTO 36),
    b   => q_12(35 DOWNTO 27),
    c   => q_13(44 DOWNTO 36),
    d   => q_12(53 DOWNTO 45),
    e   => q_11(53 DOWNTO 45),
    f   => q_11(35 DOWNTO 27),
    g   => q_13(35 DOWNTO 27),
    h   => q_13(53 DOWNTO 45),
    Lapla => L4_aux
 );

laplaciana5: laplaciana PORT MAP (
    clk => clk,
    reset => reset,
    a   => q_11(35 DOWNTO 27),
    b   => q_12(26 DOWNTO 18),
    c   => q_13(35 DOWNTO 27),
    d   => q_12(44 DOWNTO 36),
    e   => q_11(44 DOWNTO 36),
    f   => q_11(26 DOWNTO 18),
    g   => q_13(26 DOWNTO 18),
    h   => q_13(44 DOWNTO 36),
    Lapla => L5_aux
 );

laplaciana6: laplaciana PORT MAP (
    clk => clk,
    reset => reset,
    a   => q_11(26 DOWNTO 18),
    b   => q_12(17 DOWNTO  9),
    c   => q_13(26 DOWNTO 18),
    d   => q_12(35 DOWNTO 27),
    e   => q_11(35 DOWNTO 27),
    f   => q_11(17 DOWNTO  9),
    g   => q_13(17 DOWNTO  9),
    h   => q_13(35 DOWNTO 27),
    Lapla => L6_aux
```

```
   );

  laplaciana7: laplaciana PORT MAP (
      clk => clk,
      reset => reset,
      a   => q_11(17 DOWNTO  9),
      b   => q_12( 8 DOWNTO  0),
      c   => q_13(17 DOWNTO  9),
      d   => q_12(26 DOWNTO 18),
      e   => q_11(26 DOWNTO 18),
      f   => q_11( 8 DOWNTO  0),
      g   => q_13( 8 DOWNTO  0),
      h   => q_13(26 DOWNTO 18),
      Lapla => L7_aux
   );
END c;
```

A.4. Código VHDL del módulo `Ctl_opt_flow`

El módulo `Ctl_opt_flow` representa no sólo la parte de control del sistema diseñado, si no que es todo el sistema. En este módulo se integran todos los módulos expuestos anteriormente y además la etapa de control del sistema.

El código completo es mostrado a continuación:

```
LIBRARY IEEE;
LIBRARY WORK;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE work.components.all;

ENTITY Ctl_opt_flow IS
 Port(
   clk                   : IN  std_logic;
   -- System Clock
   ddr_clk               : IN  std_logic;
   Rstn                  : IN  std_logic;
   ddr_local_addr        : IN  std_logic_vector(24 downto 0);
   -- local address del controlador
   ddr_local_write_req   : IN  std_logic;
   -- write request del controlador para saber cuando hace la petición de escritura
   rd_ddr_pix            : OUT std_logic;
   -- señal que hace una petición de lectura a la memoria
   ddr_adr2              : OUT std_logic_vector(24 downto 0);
   -- dirección de la memoria que se quiere leer
   ddr_local_rdata_valid : IN  std_logic;
   -- indica que los datos leídos están disponibles
   ddr_local_rdata       : IN  std_logic_vector(63 downto 0);
   -- datos leídos de la ddram
   sdram_end_txfr        : IN  std_logic;
   -- proviene de la interfaz ddr e indica que se han terminado las operaciones RD/WR
   ddr_local_ready       : IN  std_logic
   -- indica que el controlador puede aceptar peticiones
    );
END Ctl_opt_flow;


ARCHITECTURE arch_Ctl_opt_flow OF Ctl_opt_flow IS

signal q_lapla_outa  : std_logic_vector(17 downto 0);
signal q_lapla_outb  : std_logic_vector(26 downto 0);
signal q_mlut        : std_logic_vector(7 downto 0);
signal rd_mlut, empty_mlut  : std_logic;

signal rd_lapla_outa, rd_lapla_outb, empty_lapla_outa, empty_lapla_outb : std_logic;
signal rd_grad_out,rd_lut1, empty_grad_out, empty_lut1  : std_logic;
signal q_grad_out  : std_logic_vector(215 downto 0);
signal q_lut1     : std_logic_vector(7 downto 0);

signal rd_vel_in, empty_vel_in, empty_lut2 : std_logic;
signal q_vel_in  : std_logic_vector(44 downto 0);

-- Señales usadas por Ctl_vels
signal rd_lut3, empty_lut3, clk_33 : std_logic;
signal q_lut3 : std_logic_vector(7 downto 0);

signal rdu2b, rdv2b, rdu1b, rdv1b, rdub, rdvb : std_logic;
```

```
signal wru2b, wrv2b, wru1b, wrv1b, wrub, wrvb : std_logic;
signal full_first_lapla, emptyv2  : std_logic;

signal    qu2, qu1, qu, qv2, qv1, qv  : std_logic_vector(71 downto 0);
signal datau2b, datav2b, datau1b, datav1b : std_logic_vector(71 downto 0);
signal datau b, datavb, tempub, tempvb : std_logic_vector(71 downto 0);
signal full_uv2, get_uv2b,ready_uv,  fullv2, fullv1, fullv   :  std_logic;
-- Si es uno indica que se debe llenar las fifos uv2

begin

  Clk33M: clk33
 port map
  (
    inclk0 => clk,
    c0 => clk_33
  );

  Ctl_grad_inst : Ctl_grad
   port map (
    ClkPci            => clk,
    Clk_33            => clk_33,
    ClkDdr            => ddr_clk,
    Rstn              => rstn,
    local_addr        => ddr_local_addr,
    -- local address del controlador
    local_write_req   => ddr_local_write_req,
    -- write request del controlador para saber cuando hace la petición de escritura
    RdDdrPix_o        => rd_ddr_pix,
    -- señal que hace una petición de lectura a la memoria
    DdrAdr_o          => ddr_adr2,
    -- dirección de la memoria que se quiere leer
    local_rdata_valid => ddr_local_rdata_valid,
    -- indica que los datos leídos están disponibles
    local_rdata       => ddr_local_rdata,
    DdrEndTxfr        => sdram_end_txfr,
    -- La interfaz indica que se han terminado las operaciones de lectura escritura
    DdrAck            => ddr_local_ready,
    -- El controlador puede aceptar peticiones
    rd_grad_out       => rd_grad_out,
    q_grad_out        => q_grad_out,
    empty_grad_out    => empty_grad_out,
    rd_lut1           => rd_lut1,
    q_lut1            => q_lut1,
    empty_lut1        => empty_lut1
    );

Ctl_grad_lap_inst : Ctl_grad_lap
  port map(
   ClkDdr            => ddr_clk,
   Clk_33            => clk_33,
   Rstn              => rstn,
   rd_grad_out       => rd_grad_out,
   rd_lut1           => rd_lut1,
   q_grad_out        => q_grad_out,
   empty_grad_out    => empty_grad_out,
   q_lut1            => q_lut1,
   empty_lut1        => empty_lut1,
   rd_vel_in         => rd_vel_in,
   q_vel_in          => q_vel_in,
   empty_vel_in      => empty_vel_in,
   empty_lapla_outb  => empty_lapla_outb,
   empty_lapla_outa  => empty_lapla_outa,
   rd_lapla_outb     => rd_lapla_outb,
```

```
   rd_lapla_outa     => rd_lapla_outa,
   q_lapla_outb      => q_lapla_outb,
   q_lapla_outa      => q_lapla_outa
    );

Ctl_vels_inst : Ctl_vels
   port map(
   ClkDdr             => ddr_clk,
   Clk_33             => clk_33,
   Rstn               => rstn,
   rd_vel_in          => rd_vel_in,
   q_vel_in           => q_vel_in,
   empty_vel_in       => empty_vel_in,
   rd_lut1            => rd_lut1,
   q_lut1             => q_lut1,
   rd_lut3            => rd_lut3,
   q_lut3             => q_lut3,
   empty_lut3         => empty_lut3,
   rdu2              => rdu2b,
   rdv2              => rdv2b,
   rdu1              => rdu1b,
   rdv1              => rdv1b,
   rdu               => rdub,
   rdv               => rdvb,
   wru2b              => wru2b,
   wrv2b              => wrv2b,
   wru1b              => wru1b,
   wrv1b              => wrv1b,
   wrub               => wrub,
   wrvb               => wrvb,
   datau2b            => datau2b,
   datav2b            => datav2b,
   datau1b            => datau1b,
   datav1b            => datav1b,
   dataub             => dataub,
   datavb             => datavb,
   full_uv2           => full_uv2,
   get_uv2b           => get_uv2b,
   ready_uv           => ready_uv,
   tempub             => tempub,
   tempvb             => tempvb,
   qu2                => qu2,
   qu1                => qu1,
   qu                 => qu,
   qv2                => qv2,
   qv1                => qv1,
   qv                 => qv,
   emptyv2            => emptyv2,
   fullv              => fullv,
   fullv1             => fullv1,
   fullv2             => fullv2,
   full_first_lap     => full_first_lapla,
   empty_lut2b        => empty_lut2,
   rd_mlut            => rd_mlut,
   empty_mlut         => empty_mlut,
   q_mlut             => q_mlut
 );

Ctl_lapla_inst: Ctl_lapla
  port map(
   ClkDdr             => ddr_clk,
   Clk_33             => Clk_33,
   Rstn               => Rstn,
   rd_lut3            => rd_lut3,
```

```
   q_lut3             => q_lut3,
   empty_lut3         => empty_lut3,
   emptyv2            => emptyv2,
   fullv              => fullv,
   fullv1             => fullv1,
   fullv2             => fullv2,
   rdu2               => rdu2b,
   rdv2               => rdv2b,
   rdu1               => rdu1b,
   rdv1               => rdv1b,
   rdu                => rdub,
   rdv                => rdvb,
   wru2b              => wru2b,
   wrv2b              => wrv2b,
   wru1b              => wru1b,
   wrv1b              => wrv1b,
   wrub               => wrub,
   wrvb               => wrvb,
   datau2b            => datau2b,
   datav2b            => datav2b,
   datau1b            => datau1b,
   datav1b            => datav1b,
   dataub             => dataub,
   datavb             => datavb,
   full_uv2           => full_uv2,
   get_uv2b           => get_uv2b,
   ready_uv           => ready_uv,
   tempu              => tempub,
   tempv              => tempvb,
   qu2                => qu2,
   qu1                => qu1,
   qu                 => qu,
   qv2                => qv2,
   qv1                => qv1,
   qv                 => qv,
   full_first_lapla   => full_first_lapla,
   empty_lut2         => empty_lut2,
   rd_vel_in          => rd_vel_in,
   q_vel_in           => q_vel_in,
   rd_mlut            => rd_mlut,
   empty_mlut         => empty_mlut,
   q_mlut             => q_mlut,
   empty_lapla_outb   => empty_lapla_outb,
   empty_lapla_outa   => empty_lapla_outa,
   rd_lapla_outb      => rd_lapla_outb,
   rd_lapla_outa      => rd_lapla_outa,
   q_lapla_outb       => q_lapla_outb,
   q_lapla_outa       => q_lapla_outa
);

end arch_Ctl_opt_flow;
```

A.5. Código VHDL del módulo `Ctl_grad_lap`

El módulo `ctl_grad_lap` inicia su función en el momento de recibir la señal *empty_grad_out* proveniente de la memoria `fifo_grad_out`. La señal *empty_grad_out* se activará sólo cuando la memoria `fifo_grad_out` tenga datos disponibles.

Una vez que la memoria `fifo_grad_out` posee datos para ser procesados, el módulo de control `ctl_grad_lap` multiplexará los 8 píxeles provenientes de la memoria `fifo_grad_out` a la nueva memoria `fifo_vel_in`, que almacena una sola componente para cada gradiente I_x, I_y e I_t. Lo importante de la multiplexación es que sólo serán leídas aquellas componentes de los píxeles que presentaron cambios significativos. En el caso de que ningún píxel haya presentado cambios significativos, entonces el sistema de multiplexación no escribirá ningún dato en la memoria `fifo_vel_in`. La memoria `fifo_vel_in` sólo registra aquellas componentes de los píxeles que han presentado cambios significativos y que serán los únicos procesados por el sistema, en las futuras etapas. Todo este proceso se ejecuta a 166 MHz y paralelamente al cálculo de los gradientes espacio-temporal.

El circuito de control `ctl_grad_lap`, iniciará los valores de LU y LV, cuando sea la primera iteración. Estos valores son necesarios para realizar el cálculo de las velocidades. En caso de que se esté realizando una iteración distinta a la primera, los valores de las Laplacianas y de los gradientes serán leídos de la memoria `fifo_lapla_out`. Los datos leídos que serán escritos en la memoria `fifo_vel_in` son de un tamaño de palabra de 45 bits. La memoria tiene una capacidad máxima de 256 palabras, dimensión igual a la de un renglón de las imágenes utilizadas en este trabajo. La siguiente etapa del sistema leerá de esta memoria los datos, LU, LV, I_x, I_y e I_t, para llevar a cabo el cálculo de las componentes de la velocidad.

```
LIBRARY IEEE;
LIBRARY WORK;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE work.components.all;

ENTITY Ctl_grad_lap is
  port(
   ClkDdr             : IN  std_logic;
   Clk_33             : IN  std_logic;
   Rstn               : IN  std_logic;
   rd_grad_out        : out std_logic;
   rd_lut1            : out std_logic;
   q_grad_out         : in  std_logic_vector(215 downto 0);
   empty_grad_out     : in  std_logic;
   q_lut1             : in  std_logic_vector(7 downto 0);
   empty_lut1         : in  std_logic;
```

```
   rd_vel_in          : in  std_logic;
   q_vel_in           : out std_logic_vector(44 downto 0);
   empty_vel_in       : out std_logic;
   empty_lapla_outb   : in  std_logic;
   empty_lapla_outa   : in  std_logic;
   rd_lapla_outb      : out std_logic;
   rd_lapla_outa      : out std_logic;
   q_lapla_outb       : in  std_logic_vector(26 downto 0);
   q_lapla_outa       : in  std_logic_vector(17 downto 0);
   nchanges           : out integer
   );
end Ctl_grad_lap;

ARCHITECTURE arch_Ctl_grad_lap of Ctl_grad_lap is

  signal data_vel_in,  data_vel_ina, data_vel_inb : std_logic_vector(44 downto 0);
  signal selec : std_logic;
  signal wr_vel_in, wr_vel_ina, wr_vel_inb, full_vel_in : std_logic;
  signal rd_lut1_aux, rd_lapaux_outa, rd_lapaux_outb : std_logic;
  signal nchanges_aux : integer;
  TYPE t_state_velin is (waitrd_gradout, rd_ch0, rd_ch1, rd_ch2, rd_ch3, rd_ch4,
  rd_ch5, rd_ch6, rd_ch7);
  signal state_velin : t_state_velin;
  signal nwrd_lut1 : integer;


BEGIN

 fifo_vel_in_inst:  fifo_vel_in PORT MAP(
 data    => data_vel_in,
 rdclk   => Clk_33,
 rdreq   => rd_vel_in,
 wrclk   => ClkDdr,
 wrreq   => wr_vel_in,
 q       => q_vel_in,
 rdempty => empty_vel_in,
   wrfull  => full_vel_in);

 data_vel_in <= data_vel_ina WHEN selec = '0' ELSE data_vel_inb;
 wr_vel_in   <= wr_vel_ina   WHEN selec = '0' ELSE wr_vel_inb;


 rd_lapaux_outa <= '1' WHEN selec = '1' and empty_lapla_outa = '0' and
 empty_lapla_outb = '0' and full_vel_in = '0' ELSE '0';
 rd_lapaux_outb <= '1' WHEN selec = '1' and empty_lapla_outa = '0' and
 empty_lapla_outb = '0' and full_vel_in = '0' ELSE '0';

 cont_rdlut1: PROCESS(ClkDdr, Rstn)
 begin
  if Rstn = '0' then
    nwrd_lut1  <= 0;
  elsif clkDdr = '1' and clkDdr'event then
        if rd_lut1_aux = '1' then
            nwrd_lut1 <= nwrd_lut1 + 1;
        end if;
  end if;
 end process;

 wr_velinb: PROCESS(ClkDdr, Rstn)
 begin
  if Rstn = '0' then
     wr_vel_inb   <= '0';
  elsif clkDdr = '1' and clkDdr'event then
      if (rd_lapaux_outa = '1' and rd_lapaux_outb = '1') then
```

```
         wr_vel_inb    <= '1';
       else
         wr_vel_inb    <= '0';
  end if;
 end if;
 end process;

 data_vel_inb  <= q_lapla_outb & q_lapla_outa;  -- gradientes y laplacianas

 wr_velin: PROCESS(ClkDdr, Rstn)
  begin
    if Rstn = '0' then
       rd_grad_out  <= '0';
       state_velin  <= waitrd_gradout;
       wr_vel_ina   <= '0';
       rd_lut1_aux  <= '0';
       nchanges_aux     <= 0;
    elsif clkDdr = '1' and clkDdr'event then
     case state_velin is
      when waitrd_gradout =>
  wr_vel_ina <= '0';
  if empty_grad_out = '0' then
  rd_grad_out      <= '1';
  rd_lut1_aux         <= '1';
  state_velin    <= rd_ch0;
  else
  rd_grad_out      <= '0';
  rd_lut1_aux         <= '0';
  state_velin    <= waitrd_gradout;
 end if;
  when rd_ch0 =>
 rd_grad_out <= '0';
 rd_lut1_aux  <= '0';
 if q_lut1(7) = '1' then
  nchanges_aux     <= nchanges_aux + 1;
wr_vel_ina   <= '1';
data_vel_ina <= q_grad_out(215 downto 189) & "00000000000000000";
 else
wr_vel_ina   <= '0';
end if;
state_velin <= rd_ch1;
   when rd_ch1 =>
    rd_grad_out <= '0';
rd_lut1_aux  <= '0';
if q_lut1(6) = '1' then
wr_vel_ina   <= '1';
nchanges_aux     <= nchanges_aux + 1;
data_vel_ina <= q_grad_out(188 downto 162) & "00000000000000000";
else
wr_vel_ina <= '0';
end if;
state_velin <= rd_ch2;
when rd_ch2 =>
if q_lut1(5) = '1' then
  nchanges_aux     <= nchanges_aux + 1;
wr_vel_ina   <= '1';
data_vel_ina <= q_grad_out(161 downto 135) & "00000000000000000";
else
wr_vel_ina <= '0';
end if;
state_velin <= rd_ch3;
when rd_ch3 =>
if q_lut1(4) = '1' then
  nchanges_aux     <= nchanges_aux + 1;
```

```
wr_vel_ina   <= '1';
data_vel_ina <= q_grad_out(134 downto 108) & "00000000000000000000";
else
wr_vel_ina <= '0';
end if;
state_velin <= rd_ch4;
when rd_ch4 =>
if q_lut1(3) = '1' then
  nchanges_aux     <= nchanges_aux + 1;
wr_vel_ina   <= '1';
data_vel_ina <= q_grad_out(107 downto 81) & "00000000000000000000";
else
wr_vel_ina <= '0';
end if;
state_velin <= rd_ch5;
when rd_ch5 =>
if q_lut1(2) = '1' then
  nchanges_aux     <= nchanges_aux + 1;
wr_vel_ina   <= '1';
data_vel_ina <= q_grad_out(80 downto 54) & "00000000000000000000";
else
wr_vel_ina <= '0';
end if;
state_velin <= rd_ch6;
when rd_ch6 =>
if q_lut1(1) = '1' then
  nchanges_aux     <= nchanges_aux + 1;
wr_vel_ina   <= '1';
data_vel_ina <= q_grad_out(53 downto 27) & "00000000000000000000";
else
wr_vel_ina <= '0';
end if;
state_velin <= rd_ch7;
  when rd_ch7 =>
if q_lut1(0) = '1' then
  nchanges_aux     <= nchanges_aux + 1;
wr_vel_ina   <= '1';
data_vel_ina <= q_grad_out(26 downto 0) & "00000000000000000000";
else
wr_vel_ina <= '0';
end if;
state_velin <= waitrd_gradout;
end case;
   end if;
end process;
rd_lut1       <= rd_lut1_aux;
rd_lapla_outa <= rd_lapaux_outa;
rd_lapla_outb <= rd_lapaux_outb;
nchanges      <= nchanges_aux;

end arch_Ctl_grad_lap;
```

A.6. Código VHDL del módulo `Ctl_vels`

Este circuito, llamado `ctl_vels`, inicia su proceso cuando la memoria `fifo_vel_in` contiene datos a procesar, de esta manera el módulo realiza la lectura de la memoria activando la señal *rd_vel_in*. Una vez que lee los datos, éstos son escritos en el módulo de velocidades activando la señal *start_vels* para que se inicie el procesado de la información. Una vez que ingresan los datos y después de 9 ciclos de reloj, que es la latencia del módulo `mod_vels`, se activa la señal *vels_ack* indicándole al circuito `ctl_vels` que existe datos válidos en la salida del módulo `mod_vels`. De esta manera los datos son leídos y escritos en la memoria `fifo_vel_out`, activando la señal *wr_vel_out*.

En caso de que sea la última iteración a efectuar, el módulo de control `ctl_vels` hacer una petición de escritura al módulo *Arbitro_DDR_Interfaz*. Así lee los datos de la memoria `fifo_vel_out`, activando la señal *wr_vel_out*, y los escribe en la SDRAM. También lee la LUT PRINCIPAL y escribe los datos en la SDRAM.

En caso de que no sea la última iteración, el circuito de control `ctl_vels` efectúa otra operación que consiste en realizar la reconstrucción de las imágenes que contienen las componentes de las velocidades, con el fin de efectuar el cálculo de la Laplaciana. Al momento de leer los datos de la memoria *fifo_vel_out* y escribirlos en la nueva memoria *fifo_lapla_in* deberán ser reconstruidas las imágenes de las componentes de la velocidad. Para ese propósito se utiliza la tabla LUT2. En función de los datos leídos en la LUT2 se sabrá si los datos leídos de la memoria *fifo_vel_out* se escribirán en la memoria *fifo_lapla_in* o en su caso se escribirán datos para reconstruir la imagen. Este proceso se lleva a cabo con la señal *w_data/ctl* que maneja un circuito lógico para este proceso. La memoria *fifo_lapla_in* tiene la capacidad de almacenar 6 renglones, 3 renglones por cada imagen. Por otro lado, los datos leídos de la LUT2 serán escritos en una nueva tabla LUT3.

```
LIBRARY IEEE;
LIBRARY WORK;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE work.components.all;

ENTITY Ctl_vels is
  port(
   ClkDdr             : IN  std_logic;
   Clk_33             : IN  std_logic;
   Rstn               : IN  std_logic;
   rd_vel_in          : out std_logic;
```

```
   q_vel_in          : in  std_logic_vector(44 downto 0);
   empty_vel_in      : in  std_logic;
   rd_lut1           : in std_logic;
   q_lut1            : in std_logic_vector(7 downto 0);
   rd_lut3           : in std_logic;
   q_lut3            : out std_logic_vector(7 downto 0);
   empty_lut3        : out std_logic;
   rdu2              : in std_logic;
   rdv2              : in std_logic;
   rdu1              : in std_logic;
   rdv1              : in std_logic;
   rdu               : in std_logic;
   rdv               : in std_logic;
   wru2b             : in std_logic;
   wrv2b             : in std_logic;
   wru1b             : in std_logic;
   wrv1b             : in std_logic;
   wrub              : in std_logic;
   wrvb              : in std_logic;
   datau2b           : in std_logic_vector(71 downto 0);
   datav2b           : in std_logic_vector(71 downto 0);
   datau1b           : in std_logic_vector(71 downto 0);
   datav1b           : in std_logic_vector(71 downto 0);
   datаub            : in std_logic_vector(71 downto 0);
   datavb            : in std_logic_vector(71 downto 0);
   full_uv2          : in std_logic;
   get_uv2b          : in std_logic;
   ready_uv          : out std_logic;
   tempub            : out std_logic_vector(71 downto 0);
   tempvb            : out std_logic_vector(71 downto 0);
   qu2               : out std_logic_vector(71 downto 0);
   qu1               : out std_logic_vector(71 downto 0);
   qu                : out std_logic_vector(71 downto 0);
   qv2               : out std_logic_vector(71 downto 0);
   qv1               : out std_logic_vector(71 downto 0);
   qv                : out std_logic_vector(71 downto 0);
   emptyv2           : out std_logic;
   fullv             : out std_logic;
   fullv1            : out std_logic;
   fullv2            : out std_logic;
   full_first_lap    : out std_logic;
   empty_lut2b       : out std_logic;
   rd_mlut           : out std_logic;
   empty_mlut        : in std_logic;
   q_mlut            : in std_logic_vector(7 downto 0);
   end_process       : out std_logic;
   nchanges          : in integer;
   vel_addr_o        : out std_logic_vector(24 downto 0);
   vel_be_o          : out std_logic_vector(7 downto 0);
   vel_data_o        : out std_logic_vector(63 downto 0);
   vel_wr_req_o      : out std_logic;
   ddr_local_ready   : in std_logic;
   local_wdata_req   : in std_logic
);
END Ctl_vels;

ARCHITECTURE arch_Ctl_vels of Ctl_vels is

  signal data_vel_out, q_vel_out   : std_logic_vector(17 downto 0);
  signal rd_vel_outa, rd_vel_outb, empty_vel_out: std_logic;
  signal wr_vel_out,  full_vel_out, rd_vel_out : std_logic;
  signal rdusedw_vel_out, wrusedw_vel_out : std_logic_vector(7 downto 0);
  signal rd_mlut_aux  : std_logic;
  signal vel_be_temp1, vel_be_temp2  : std_logic_vector(7 downto 0);
```

```
  signal vel_be, vel_be_temp  : std_logic_vector(7 downto 0);
  signal U, V   : std_logic_vector(8 DOWNTO 0);
  signal vels_start, vels_ack, clr  : std_logic;
  signal data_lut2, data_lut2a, data_lut2b, q_lut2  : std_logic_vector(7 downto 0);
  signal full_lut2, rd_lut2, rd_lut2a, rd_lut2b, empty_lut2 : std_logic;
  signal wr_lut2, wr_lut2a, wr_lut2b, wr_lut2_selec  : std_logic;
  signal usedw_lut2  : std_logic_vector(10 downto 0);
  signal datav, tempu, tempv, qu2a, qu1a   : std_logic_vector(71 downto 0);
  signal qua, qv2a, qv1a, qva, dataua, datava : std_logic_vector(71 downto 0);
  signal datau2, datav2, datau1, datav1, datau  : std_logic_vector(71 downto 0);
  signal datau2a, datav2a, datau1a, datav1a   : std_logic_vector(71 downto 0);
  signal wru2,  emptyu2, fullu2 : std_logic;
  signal wru1,  emptyu1, fullu1 : std_logic;
  signal wru,   emptyu,  fullu  : std_logic;
  signal wrv2    : std_logic;
  signal wrv1,  emptyv1  : std_logic;
  signal wrv,   emptyv   : std_logic;
  signal ready_uv_aux  : std_logic;
  signal data_lut3  : std_logic_vector(7 downto 0);
  signal wr_lut3, full_lut3  : std_logic;
  signal usedw_lut3          : std_logic_vector(10 downto 0);
  signal inc_addr  : std_logic;

  TYPE t_state_rdfifovels IS (wait_rdvels, read_vels, get_vels);
  TYPE t_state_loaduv2 IS (wait_worduv2, write_uvn2,idle2);
  TYPE t_state_loaduv1 IS (wait_worduv1, write_uvn1, idle1);
  TYPE t_state_loaduv  IS (wait_worduv,  write_uvn, idle);
  TYPE t_state_getuv   IS (wait_getuv, wait_datalut2,read_ch1, rd_vel1,
  read_ch2, rd_vel2, read_ch3, rd_vel3, read_ch4, rd_vel4, read_ch5,
  rd_vel5, read_ch6, rd_vel6, read_ch7, rd_vel7, read_ch8, rd_vel8);

  signal state_getuv      : t_state_getuv;
  signal state_loaduv2    : t_state_loaduv2;
  signal state_loaduv1    : t_state_loaduv1;
  signal state_loaduv     : t_state_loaduv;
  signal state_rdfifovels  : t_state_rdfifovels;
  signal get_uv,get_uv1, get_uv2, read_fifovels,ready_vel  : std_logic;
  signal wru2a, wrv2a, wru1a, wrv1a, wrua, wrva  : std_logic;
  signal rdusedwu2, wrusedwu2, rdusedwv2, wrusedwv2 : STD_LOGIC_VECTOR (4 DOWNTO 0);
  signal clk_fifo_lapla, swddr, sw33  : std_logic;
  signal clk_fifo_lapla2, swddr2, sw332  : std_logic;
  signal full_first_lapla : std_logic;
  signal nwr_lut2, nwr_lut2a  : integer;
  signal nrd_velin, nrd_lut2b, nrd_veloutb  : integer;
  signal niteracion  : integer;
  signal niteraciones  : integer;
  signal end_proc     : std_logic;  -- fin del calculo del flujo optico
  signal vel_wr_req, vel_wr_reqa, vel_wr_reqb, vel_wr_reqc : std_logic;
  signal vel_addr  : std_logic_vector(24 downto 0);
  -- direccion dodnde se escribe el resultado de velocidades
  signal vel_temp, vel_temp1  : std_logic_vector(63 downto 0);


BEGIN

 fifo_vel_out_inst:  fifo_vel_out PORT MAP(
 data    => data_vel_out,
 rdclk   => ClkDdr,
 rdreq   => rd_vel_out,
 wrclk   => Clk_33,
 wrreq   => wr_vel_out,
 q       => q_vel_out,
 rdempty => empty_vel_out,
 rdusedw => rdusedw_vel_out,
```

```
 wrfull  => full_vel_out,
 wrusedw => wrusedw_vel_out
 );

 vel_inst: velocidad PORT MAP(
clk         => Clk_33,
reset       => Rstn,
vels_start => vels_start,
Ix          => q_vel_in(44 downto 36),
Iy          => q_vel_in(35 downto 27),
It          => q_vel_in(26 downto 18),
LU          => q_vel_in(17 downto 9),
LV          => q_vel_in(8 downto 0),
vels_ack => vels_ack,
U           => U,
V           => V
  );

  lut2_inst: fifo_lutsclk PORT MAP(
    clock    => ClkDdr,
data  => data_lut2,
rdreq    => rd_lut2,
wrreq    => wr_lut2,
empty  => empty_lut2,
full     => full_lut2,
q      => q_lut2,
usedw  => usedw_lut2
);

  un2: fifo_lapla_in1 PORT MAP(
     aclr    => clr,
     data    => datau2,
     wrreq  => wru2,
     rdreq  => rdu2,
     clock  => clk_fifo_lapla2,
     q      => qu2a,
     full    => fullu2,
     empty  => emptyu2
   );

  un1: fifo_lapla_in1 PORT MAP(
     aclr    => clr,
     data    => datau1,
     wrreq  => wru1,
     rdreq  => rdu1,
     clock  => clk_fifo_lapla,
     q      => qu1a,
     full    => fullu1,
     empty  => emptyu1
   );

  un: fifo_lapla_in1 PORT MAP(
     aclr    => clr,
     data    => datau,
     wrreq  => wru,
     rdreq  => rdu,
     clock  => clk_fifo_lapla,
     q      => qua,
     full    => fullu,
     empty  => emptyu
   );

  vn2: fifo_lapla_in1 PORT MAP(
     aclr    => clr,
```

```
     data    => datav2,
     wrreq  => wrv2,
     rdreq  => rdv2,
     clock  => clk_fifo_lapla2,
     q      => qv2a,
     full    => fullv2,
     empty  => emptyv2
   );

  vn1: fifo_lapla_in1 PORT MAP(
     aclr    => clr,
     data    => datav1,
     wrreq  => wrv1,
     rdreq  => rdv1,
     clock   => clk_fifo_lapla,
     q      => qv1a,
     full    => fullv1,
     empty  => emptyv1
   );

  vn: fifo_lapla_in1 PORT MAP(
     aclr   => clr,
     data   => datav,
     wrreq  => wrv,
     rdreq  => rdv,
     clock  => clk_fifo_lapla,
     q      => qva,
     full   => fullv,
     empty => emptyv
   );

  lut3_inst: fifo_lutsclk PORT MAP(
    clock    => ClkDdr,
data  => data_lut3,
rdreq    => rd_lut3,
wrreq    => wr_lut3,
empty  => empty_lut3,
full     => full_lut3,
q      => q_lut3,
usedw  => usedw_lut3
 );

  -- Multiplexor para escribir la lut2
  wr_lut2    <= wr_lut2a   WHEN wr_lut2_selec = '0' ELSE wr_lut2b;
  data_lut2  <= data_lut2a WHEN wr_lut2_selec = '0' ELSE data_lut2b;



  -- escribe en la lut2, los datos leidos de la main_lut
  data_lut2b  <= q_mlut;

  rd_mlut_aux <= '1' WHEN wr_lut2_selec = '1' and empty_mlut = '0'  and
  full_lut2 = '0' ELSE '0';

  write_lut2b: PROCESS(ClkDdr, Rstn)
  begin
     if Rstn = '0' then
        wr_lut2b <= '0';
     elsif clkDdr = '1' and clkDdr'event then
        if rd_mlut_aux = '1' then
           wr_lut2b <= '1';
        else
           wr_lut2b <= '0';
        end if;
```

```
    end if;
 end process;

 -- escribe en la lut2, los datos leídos de la lut1
 data_lut2a <= q_lut1;

 write_lut2a: PROCESS(ClkDdr, Rstn)
 begin
    if Rstn = '0' then
       nwr_lut2a  <=  0;
       wr_lut2a   <= '0';
    elsif ClkDdr = '1' AND ClkDdr'event THEN
       if rd_lut1 = '1' then
          nwr_lut2a <= nwr_lut2a + 1;
          wr_lut2a  <= '1';
       else
          wr_lut2a <= '0';
       end if;
   end if;
end process;

 data_lut3 <= q_lut2;

 wr_lut3_p: PROCESS(ClkDdr, Rstn)
 begin
    if Rstn = '0' then
       wr_lut3   <= '0';
    elsif ClkDdr = '1' AND ClkDdr'event THEN
       if rd_lut2a = '1' then
          wr_lut3 <= '1';
       else
          wr_lut3 <= '0';
       end if;
   end if;
 end process;

 PROCESS(Clk_33, Rstn)
 BEGIN
     IF Rstn = '0' THEN
        vels_start <= '0';
     ELSIF (Clk_33 = '1' AND Clk_33'event)THEN
        IF empty_vel_in = '0' THEN
         vels_start <= '1';
        ELSE
           vels_start <= '0';
        END IF;
    END IF;
 END PROCESS;

 rd_vel_in <= NOT empty_vel_in;

 PROCESS(Clk_33, Rstn)
 BEGIN
    IF Rstn = '0' THEN
      wr_vel_out <= '0';
    ELSIF Clk_33 = '1' AND Clk_33'event THEN
      IF vels_ack = '1' THEN
        wr_vel_out <=  '1';
        data_vel_out(17 DOWNTO 9) <= U;
        data_vel_out(8 DOWNTO 0)  <= V;
     ELSE
       wr_vel_out  <= '0';
     END IF;
   END IF;
```

```
END PROCESS;

-- contador del numero de iteraciones
process(Rstn, full_first_lapla)
begin
   if Rstn = '0'   then
     niteracion  <= 1;
   elsif full_first_lapla = '0'  THEN
     niteracion <= niteracion + 1;
   end if;
end process;

-- Termina cuado se cumple el numero de iteraciones
wr_end_proc: PROCESS(ClkDdr, Rstn)
begin
   if Rstn = '0' then
     end_proc   <= '0';
   elsif ClkDdr = '1' AND ClkDdr'event THEN
      if niteracion < niteraciones then
         end_proc <= '0';
      else
         end_proc <= '1';
      end if;
  end if;
end process;

--- ESCRITURA EN LA DDR-SDRAM

-- Hay palabras en la lut2 y velocidades por escribir
vel_wr_reqa  <= '1' when empty_lut2 = '0' and empty_vel_out = '0'    else '0';
-- Hay palabras en la lut2 pero no hay velocidades por escribir
vel_wr_reqb  <= '1' when empty_lut2 = '0' and nrd_veloutb = nchanges else '0';

-- Hay velocidades pero no hay palabras en la lut2
vel_wr_reqc  <= '1' when empty_vel_out = '0' and (empty_lut2 = '1' and
nrd_lut2b = 128) else '0';

vel_wr_req   <= '1' when (vel_wr_reqa = '1' or vel_wr_reqb = '1' or
vel_wr_reqc = '1') and end_proc = '1' and ddr_local_ready = '1' else '0';

rd_lut2b     <= '1'  when empty_lut2     =  '0'  and vel_wr_req = '1' and
end_proc = '1' else '0';
rd_vel_outb  <= '1'  when empty_vel_out  =  '0'  and vel_wr_req = '1' and
end_proc = '1' else '0';

process(Rstn, clkddr)
begin
   if Rstn =  '0' then
     vel_addr       <= "00000000001000000000000000";  --0004000
   elsif ClkDdr = '1' AND ClkDdr'event then
      if inc_addr = '1' and vel_wr_req = '1' then
         vel_addr     <= vel_addr + 1;
      end if;
  end if;
end process;

process(Rstn, clkddr)
begin
   if Rstn =  '0' then
     vel_be_temp        <= "11110000";
   elsif ClkDdr = '1' AND ClkDdr'event then
      if rd_lut2b = '1' or rd_vel_outb = '1' then
         vel_be_temp  <=  not vel_be_temp;
      end if;
```

```
   end if;
end process;

process(Rstn, clkddr)
begin
   if Rstn =  '0' then
      vel_be_temp1       <= "11111111";
   elsif ClkDdr = '1' AND ClkDdr'event then
      vel_be_temp1 <=  vel_be_temp;
      vel_be       <=  vel_be_temp1;
   end if;
end process;

process(Rstn, clkddr)
begin
   if Rstn =  '0' then
      inc_addr  <= '0';
   elsif ClkDdr = '1' AND ClkDdr'event then
       if vel_wr_req = '1'  then
          inc_addr    <=  not inc_addr;
       end if;
  end if;
end process;

process(ClkDdr)
begin
   if ClkDdr = '1' AND ClkDdr'event then
     vel_temp   <= q_lut2 & q_vel_out & "000000" & q_lut2 & q_vel_out & "000000";
     vel_data_o <= vel_temp;
   end if;
end process;

PROCESS(ClkDdr, Rstn)
BEGIN
   IF Rstn = '0' THEN
      nrd_veloutb <= 0;
   ELSIF ClkDdr = '1' AND ClkDdr'event THEN
     IF rd_vel_outb = '1' THEN
        nrd_veloutb <= nrd_veloutb + 1;
     END IF;
  END IF;
END PROCESS;

PROCESS(ClkDdr, Rstn)
BEGIN
   IF Rstn = '0' THEN
      nrd_lut2b <= 0;
   ELSIF ClkDdr = '1' AND ClkDdr'event THEN
      IF rd_lut2b = '1' THEN
         nrd_lut2b <= nrd_lut2b + 1;
      END IF;
  END IF;
END PROCESS;

load_uv: PROCESS( ClkDdr, Rstn)
BEGIN
  IF Rstn = '0' THEN
   get_uv       <= '0';
   -- Señal que activa la lectura de la lut2 y ordena las velocidades
   wrua        <= '0';
   wrva        <= '0';
  ELSIF ClkDdr = '1' AND ClkDdr'event and full_first_lapla= '0' and
   end_proc = '0' THEN
    CASE state_loaduv IS
```

```
        WHEN wait_worduv =>
                          get_uv <= '0';
                          wrua   <= '0';
                          wrva   <= '0';
                          IF empty_lut2 = '0' and fullu = '0' THEN
                             get_uv  <= '1';
                             state_loaduv <= write_uvn;
                          END IF;
        WHEN write_uvn =>
                          get_uv <= '0';
                          IF ready_uv_aux = '1' THEN
                             dataua <= tempu;
                             datava <= tempv;
                         wrua   <= '1';
                             wrva   <= '1';
                             state_loaduv <= idle ;
                          END IF;
         WHEN idle   =>
                          get_uv <= '0';
                          wrua   <= '0';
                          wrva   <= '0';
                          state_loaduv <= wait_worduv;
    END CASE;
  END IF;
END PROCESS;

load_uv1: PROCESS( ClkDdr, Rstn)
BEGIN
  IF Rstn = '0' THEN
     get_uv1       <= '0';
     wru1a        <= '0';
     wrv1a        <= '0';
  ELSIF ClkDdr = '1' AND ClkDdr'event and full_first_lapla= '0' and
  end_proc = '0' THEN
    CASE state_loaduv1 IS
       WHEN wait_worduv1 =>
                    get_uv1 <= '0';
                    wru1a   <= '0';
                    wrv1a   <= '0';
                    IF empty_lut2 = '0' and fullu1 = '0' and fullu = '1' THEN
                       get_uv1  <= '1';
                       state_loaduv1 <= write_uvn1;
                    END IF;
      WHEN write_uvn1 =>
                    get_uv1 <= '0';
                    IF ready_uv_aux = '1' THEN
                       datau1a <= tempu;
                       datav1a <= tempv;
                       wru1a  <= '1';
                       wrv1a  <= '1';
                       state_loaduv1 <= idle1;
                    END IF;
      WHEN idle1 =>
                    get_uv1 <= '0';
                    wru1a   <= '0';
                    wrv1a   <= '0';
                    state_loaduv1 <= wait_worduv1;
    END CASE;
  END IF;
END PROCESS;

load_uv2: PROCESS( ClkDdr, Rstn)
BEGIN
  IF Rstn = '0' THEN
```

```
     get_uv2       <= '0';
     -- Señal que activa la lectura de la lut2 y ordena las velocidades
     wru2a        <= '0';
     wrv2a        <= '0';
 ELSIF ClkDdr = '1' AND ClkDdr'event and full_first_lapla= '0' and end_proc = '0' THEN
  CASE state_loaduv2 IS
       WHEN wait_worduv2 =>
                         get_uv2 <= '0';
                         wru2a   <= '0';
                         wrv2a   <= '0';
                         IF empty_lut2 = '0' and fullu2 = '0' and fullu1 = '1' THEN
                            get_uv2  <= '1';
                            state_loaduv2 <= write_uvn2;
                         END IF;
       WHEN write_uvn2 =>
                         get_uv2 <= '0';
                         IF ready_uv_aux = '1' THEN
                            datau2a <= tempu;
                            datav2a <= tempv;
                          wru2a  <= '1';
                            wrv2a  <= '1';
                            state_loaduv2 <= idle2;
                         END IF;
      WHEN idle2 =>
                         get_uv2 <= '0';
                         wru2a   <= '0';
                         wrv2a   <= '0';
                         state_loaduv2 <= wait_worduv2;
  END CASE;
 END IF;
END PROCESS;


-- Multiplexor para leer la fifo de velocidades y lut2 que cambiaron,
-- la lectura puede ser para continuar las iteraciones o escribir el resultado final
rd_vel_out <= rd_vel_outa when end_proc = '0' else rd_vel_outb;
rd_lut2    <= rd_lut2a    when end_proc = '0' else rd_lut2b;

rd_fifovel_out: PROCESS(clkDdr, Rstn)
BEGIN
  IF Rstn = '0' THEN
     ready_vel        <= '0';
     rd_vel_outa       <= '0';
     state_rdfifovels <= wait_rdvels;
  ELSIF clkDdr= '1' AND clkDdr'event and end_proc = '0' THEN
    CASE state_rdfifovels IS
        WHEN wait_rdvels =>
                           rd_vel_outa <= '0';
                           ready_vel  <= '0';
                           -- Señal que indica que se tiene la velocidad
                           IF read_fifovels = '1' AND  empty_vel_out = '0' THEN
                              state_rdfifovels <= read_vels;
                           END IF;
        WHEN read_vels   =>
                           rd_vel_outa        <= '1';
                           state_rdfifovels <= get_vels;
        WHEN get_vels    =>
                           ready_vel <= '1';
                           rd_vel_outa <= '0';
                           state_rdfifovels <= wait_rdvels;
    END CASE;
  END IF;
END PROCESS;
```

```
PROCESS(Clk_33, Rstn)
BEGIN
  IF Rstn = '0' THEN
     nrd_velin <= 0;
  ELSIF Clk_33 = '1' AND Clk_33'event THEN
    IF rdu2 = '1' THEN
     nrd_velin <= nrd_velin + 1;
    elsif nrd_velin = (nren - 3) * 32 then --(nrenglones - 3) * 32
     nrd_velin <= 0;
    END IF;
 END IF;
END PROCESS;

PROCESS(Rstn, nrd_velin)
BEGIN
  IF Rstn = '0' THEN
    clr  <= '0';
  elsif nrd_velin = (nren - 3) * 32 then -- nren-3 * 32
    clr <= '1';
  else
    clr <= '0';
  end if;
end process;

--Start_lapmod marca los siguientes dos casos:
--full_first_lapla = 0, llenado de las tres fifos de velocidades por primera vez.
--full_firts_lapla = 1, lectura de las tres fifos y corrimiento de renglones para
--el calculo de la laplaciana.

-- Indica que se han llenado las seis fifos de las laplacianas por lo que se
--pueden empezar a leer las fifos
process(Rstn, fullu1, fullu2, fullu, emptyu2, emptyu1, emptyu)
begin
    if (Rstn = '0') then
       full_first_lapla <= '0';
    elsif fullu2 = '1' and fullu1 = '1' and fullu = '1' then
      full_first_lapla <= '1';
   elsif   emptyu2 = '1' and emptyu1 = '1' and emptyu = '1' then --new
     full_first_lapla <= '0';  --new
  end if;
end process;

-- Cuando se han llenado las seis fifos se cambia la fwrite de las fifos u,
--u1, v y v1 a 33 mhz
process(Rstn,  full_first_lapla)
begin
 if (Rstn = '0') then
    swddr   <= '1';
    sw33    <= '0';
 elsif full_first_lapla = '1' then
    swddr   <= '0';
    sw33    <= '1';
 else
    swddr   <= '1';
    sw33    <= '0';
 end if;
end process;

 -- Cuando se han llenado las seis fifos se cambia la fwrite de las fifos
 --u, u1, v y v1 a 33 mhz
process(Rstn,  full_first_lapla, full_uv2)
begin
 if (Rstn = '0') then
```

```
     swddr2   <= '1';
     sw332   <= '0';
 elsif full_first_lapla = '0' or full_uv2 = '1' then
     swddr2   <= '1';
     sw332    <= '0';
 else
    swddr2   <= '0';
    sw332    <= '1';
 end if;
end process;

get_datauv: PROCESS(ClkDdr, Rstn)
BEGIN
  IF Rstn = '0' THEN
   read_fifovels <= '0';
   state_getuv  <= wait_getuv;
   ready_uv_aux     <= '0';
   rd_lut2a       <= '0';
ELSIF (ClkDdr = '1' AND ClkDdr'event) THEN
  CASE state_getuv IS
       WHEN wait_getuv =>
                          read_fifovels <= '0';
                          ready_uv_aux  <= '0';
                       IF get_uv = '1' or get_uv1 = '1' or get_uv2 = '1' or
                         get_uv2b = '1' THEN  --  activa la lectura de la lut2
                            rd_lut2a      <= '1';
                            state_getuv <= wait_datalut2;
                       END IF;
       WHEN wait_datalut2 =>
                            rd_lut2a      <= '0';
                            state_getuv <= read_ch1;
       WHEN read_ch1    =>
                            IF q_lut2(7 DOWNTO 7) = "1" THEN
                               read_fifovels <= '1';
                               state_getuv  <= rd_vel1;
                            ELSE
                             tempu(71 DOWNTO 63) <= (OTHERS => '0');
                             tempv(71 DOWNTO 63) <= (OTHERS => '0');
                             state_getuv <= read_ch2;
                            END IF;
       WHEN rd_vel1     =>
                           IF state_rdfifovels = read_vels THEN
                               read_fifovels <= '0';
                           END IF;
                           IF ready_vel = '1' THEN
                      tempu(71 DOWNTO 63) <= q_vel_out(17 DOWNTO 9);
                              tempv(71 DOWNTO 63) <= q_vel_out(8 DOWNTO 0);
                              state_getuv <= read_ch2;
                           END IF;

       WHEN read_ch2    =>
                            IF q_lut2(6 DOWNTO 6) = "1" THEN
                               read_fifovels <= '1';
                               state_getuv  <= rd_vel2;
                            ELSE
                             tempu(62 DOWNTO 54) <= (OTHERS => '0');
                             tempv(62 DOWNTO 54) <= (OTHERS => '0');
                             state_getuv <= read_ch3;
                            END IF;
       WHEN rd_vel2     =>
                           IF state_rdfifovels = read_vels THEN
                             read_fifovels <= '0';
                           END IF;
                           IF ready_vel = '1' THEN
```

```
                tempu(62 DOWNTO 54) <= q_vel_out(17 DOWNTO 9);
                       tempv(62 DOWNTO 54) <= q_vel_out(8  DOWNTO 0);
                       state_getuv <= read_ch3;
                    END IF;
WHEN read_ch3    =>
                      IF q_lut2(5 DOWNTO 5) = "1" THEN
                         read_fifovels <= '1';
                         state_getuv <= rd_vel3;
                      ELSE
                       tempu(53 DOWNTO 45) <= (OTHERS => '0');
                       tempv(53 DOWNTO 45) <= (OTHERS => '0');
                       state_getuv <= read_ch4;
                      END IF;
WHEN rd_vel3     =>
                     IF state_rdfifovels = read_vels THEN
                       read_fifovels <= '0';
                     END IF;
                IF ready_vel = '1' THEN
                tempu(53 DOWNTO 45) <= q_vel_out(17 DOWNTO 9);
                       tempv(53 DOWNTO 45) <= q_vel_out(8 DOWNTO 0);
                       state_getuv <= read_ch4;
                     END IF;
WHEN read_ch4    =>
                      IF q_lut2(4 DOWNTO 4) = "1" THEN
                          read_fifovels <= '1';
                          state_getuv <= rd_vel4;
                      ELSE
                          tempu(44 DOWNTO 36) <= (OTHERS => '0');
                          tempv(44 DOWNTO 36) <= (OTHERS => '0');
                          state_getuv <= read_ch5;
                      END IF;
WHEN rd_vel4     =>
                      IF state_rdfifovels = read_vels THEN
                         read_fifovels <= '0';
                      END IF;
                    IF ready_vel = '1' THEN
                   tempu(44 DOWNTO 36) <= q_vel_out(17 DOWNTO 9);
                         tempv(44 DOWNTO 36) <= q_vel_out(8  DOWNTO 0);
                         state_getuv <= read_ch5;
                      END IF;
WHEN read_ch5    =>
                      IF q_lut2(3 DOWNTO 3) = "1" THEN
                         read_fifovels <= '1';
                         state_getuv <= rd_vel5;
                      ELSE
                         tempu(35 DOWNTO 27) <= (OTHERS => '0');
                         tempv(35 DOWNTO 27) <= (OTHERS => '0');
                         state_getuv <= read_ch6;
                      END IF;
WHEN rd_vel5     =>
                      IF state_rdfifovels = read_vels THEN
                         read_fifovels <= '0';
                      END IF;
                IF ready_vel = '1' THEN
                 tempu(35 DOWNTO 27) <= q_vel_out(17 DOWNTO 9);
                         tempv(35 DOWNTO 27) <= q_vel_out(8 DOWNTO 0);
                         state_getuv <= read_ch6;
                      END IF;
WHEN read_ch6    =>
                      IF q_lut2(2 DOWNTO 2) = "1" THEN
                         read_fifovels <= '1';
                         state_getuv <= rd_vel6;
                      ELSE
                         tempu(26 DOWNTO 18) <= (OTHERS => '0');
```

```
                              tempv(26 DOWNTO 18) <= (OTHERS => '0');
                              state_getuv <= read_ch7;
                            END IF;
          WHEN rd_vel6     =>
                            IF state_rdfifovels = read_vels THEN
                              read_fifovels <= '0';
                            END IF;
                          IF ready_vel = '1' THEN
                           tempu(26 DOWNTO 18) <= q_vel_out(17 DOWNTO 9);
                              tempv(26 DOWNTO 18) <= q_vel_out(8  DOWNTO 0);
                              state_getuv <= read_ch7;
                            END IF;
          WHEN read_ch7    =>
                            IF q_lut2(1 DOWNTO 1) = "1" THEN
                              read_fifovels <= '1';
                              state_getuv <= rd_vel7;
                            ELSE
                              tempu(17 DOWNTO 9) <= (OTHERS => '0');
                              tempv(17 DOWNTO 9) <= (OTHERS => '0');
                              state_getuv <= read_ch8;
                            END IF;
          WHEN rd_vel7     =>
                            IF state_rdfifovels = read_vels THEN
                              read_fifovels <= '0';
                            END IF;
                          IF ready_vel = '1' THEN
                           tempu(17 DOWNTO 9) <= q_vel_out(17 DOWNTO 9);
                              tempv(17 DOWNTO 9) <= q_vel_out(8  DOWNTO 0);
                              state_getuv <= read_ch8;
                            END IF;
          WHEN read_ch8    =>
                            IF q_lut2(0 DOWNTO 0) = "1" THEN
                              read_fifovels <= '1';
                              state_getuv  <= rd_vel8;
                            ELSE
                              tempu(8 DOWNTO 0) <= (OTHERS => '0');
                              tempv(8 DOWNTO 0) <= (OTHERS => '0');
                              state_getuv <= wait_getuv;
                              ready_uv_aux    <= '1';
                            END IF;
          WHEN rd_vel8     =>
                            IF state_rdfifovels = read_vels THEN
                              read_fifovels <= '0';
                            END IF;
                            IF ready_vel = '1' THEN
                           tempu(8 DOWNTO 0) <= q_vel_out(17 DOWNTO 9);
                              tempv(8 DOWNTO 0) <= q_vel_out(8  DOWNTO 0);
                              state_getuv <= wait_getuv;
                              ready_uv_aux    <= '1';
                            END IF;
     END CASE;
  END IF;
END PROCESS;

niteraciones <= 4;
ready_uv <= ready_uv_aux;
wru2 <= wru2a OR wru2b;
wrv2 <= wrv2a OR wrv2b;
wru1 <= wru1a OR wru1b;
wrv1 <= wrv1a OR wrv1b;
wru  <= wrua  OR wrub;
wrv  <= wrva  OR wrvb;

tempub <= tempu;
```

```
tempvb <= tempv;

datau2 <= datau2a or datau2b;
datau1 <= datau1a or datau1b;
datau  <= dataua  or datab;
datav2 <= datav2a or datav2b;
datav1 <= datav1a or datav1b;
datav  <= datava  or datavb;


clk_fifo_lapla  <= (swddr and ClkDdr) or (sw33 and Clk_33);
clk_fifo_lapla2 <= (swddr2 and ClkDdr) or (sw332 and Clk_33);

qu2            <= qu2a;
qu1            <= qu1a;
qu             <= qua;
qv2            <= qv2a;
qv1            <= qv1a;
qv             <= qva;
full_first_lap <= full_first_lapla;
empty_lut2b    <= empty_lut2;
rd_mlut        <= rd_mlut_aux;
end_process    <= end_proc;

-- señales para petición de escitura en ddr
vel_addr_o     <= vel_addr;
vel_be_o       <= vel_be;
vel_wr_req_o   <= vel_wr_req;

 end arch_Ctl_vels;
```

Apéndice B

Bus PCI

Los ordenadores personales (PC's) se concibieron como computadores con funcionalidades de reconfiguración que disponen de conectores para instalar más memoria o tarjetas de expansión que implementan funciones avanzadas o no disponibles en el computador básico. Todos los componentes de un computador están interconectados entre sí e intercambian información mediante un BUS. De tal manera que dependiendo del bus se consiguen mejores o peores prestaciones del ordenador. El bus PCI (Peripheral Component Interconnect) fue creado por Intel con la idea fundamental de definir un bus de alta velocidad independiente del bus del procesador (Bus Local Residente).

Intel cedió su patente al dominio público y promovió la creación de una asociación industrial, la PCI-SIG (PCI - Special Interest Group)[1], para continuar el desarrollo y mantener la compatibilidad de las especificaciones del bus PCI. El resultado ha sido que el bus PCI ha sido ampliamente adoptado.

La primera especificación del bus PCI versión 1.0 fue presentada por Intel en el año de 1992. Posteriormente el PCI-SIG ha presentado nuevas revisiones que hasta la fecha está en la revisión 2,3. La revisión usada en este trabajo es la 2,2, compatible con el sistema utilizado. Cabe mencionar que además de la nueva revisión también hay una nueva generación del bus PCI, como el PCI-X o el PCI-Express.

Las tarjetas controladoras de periféricos diseñadas para el bus local PCI tienen especificaciones de autoconfiguración grabadas en una memoria incluida en la misma tarjeta, para proveer la información de instalación necesaria para el sistema durante la fase de arranque. Las rutinas de la BIOS configuran automáticamente

[1] http://www.pci.org

cada dispositivo de PCI basándose en los recursos que ya están en uso por otras tarjetas. Teóricamente, el usuario no tiene que ajustar interruptores o puentes para elegir niveles de IRQ, de DMA o direcciones de memoria cada vez que agregue un periférico al sistema.

En este apéndice se ilustran brevemente las características y conceptos generales del bus PCI. Explicándose los mecanismos de transacción de lectura y escritura de los registros de configuración, de memoria y de I/O.

B.1. Características del Bus

Como se mencionó anteriormente existen varias revisiones del bus PCI. La revisión usada en este trabajo es la revisión 2,2 compatible con la plataforma de desarrollo utilizada, la *Stratix PCI development board*. Es posible considerar que el bus PCI es un bus paralelo, multiplexado y síncrono. Existe una gran cantidad de características, pero las más destacadas son:

- Las tarjetas controladoras de periféricos diseñadas para el bus local PCI tienen especificaciones de autoconfiguración. Éstas están grabadas en una memoria incluida en la misma tarjeta. Así se proveer la información de instalación necesaria para el sistema, durante la fase de arranque. Las rutinas de la BIOS configuran automáticamente cada dispositivo de PCI basándose en los recursos que ya están en uso, por otras tarjetas.
- Frecuencia de operación de 33 MHz y 66 MHz.
- Velocidades de transferencia de 33 MHz * 32 bits = 132 MB/s. Se permite una extensión transparente de 64 bits, del bus de datos, que a una frecuencia de 66 MHz darían lugar a 528 MB/s.
- Independiente del procesador. Idealmente se garantiza una transición de los dispositivos a futuras generaciones de procesadores.
- Configuración automática. Mediante un espacio de configuración compuesto por una colección de registros propio de cada dispositivo ("Plug&Play").
- Capacidad de soportar varias tarjetas operando concurrentemente sobre el bus. Pudiendo soportar el modo de operación "multimaster".
- Integridad de datos. El protocolo del bus PCI proporciona control de paridad y chequeo de errores durante las transferencias de datos y direcciones.

B.2. Estructura del Bus

La interfaz PCI requiere un mínimo de 47 señales para operar como esclavo(*target o slave*) ó 49 para operar como maestro (*iniciator o master*) y pueda manejar datos y direcciones, control del interface, arbitraje y funciones del sistema. La figura B.1 muestra las señales en grupos funcionales. A la izquierda de la figura se muestran las líneas necesarias y a la derecha las opcionales. Los sentidos de las flechas son para un dispositivo master-target. Las líneas pueden ser de distinto tipo, las de entrada son estándar, las de salida son tipo *totem pole*, otras son triestado, triestado sostenido y las hay también de drenador abierto. Las líneas que tengan un # al final del nombre corresponde a señales que son activas en nivel bajo.

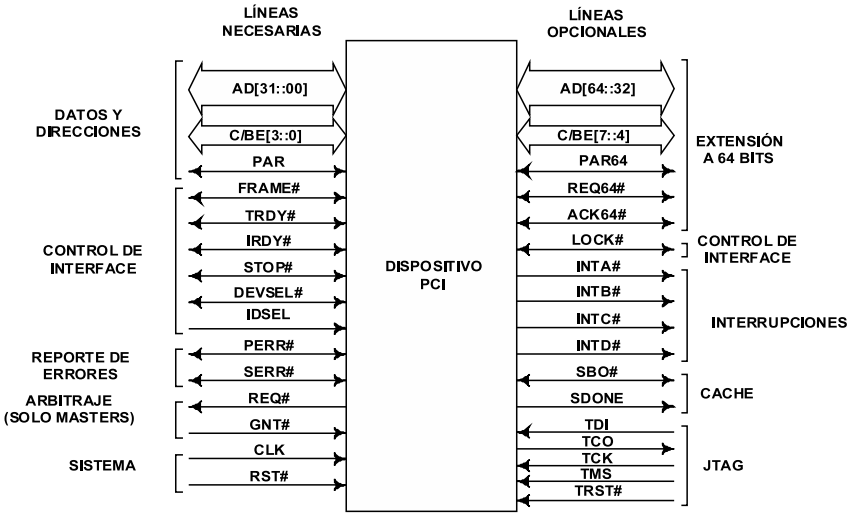


Figura B.1: *Diagrama a bloques de las tres etapas de simulación y de cada uno de los módulos que las conforman.*

Las terminales obligatorias necesarias para que el bus PCI trabaje en modo maestro son 49. Estas terminales se pueden dividir en los siguientes grupos funcionales:

1. **Terminales de sistema:** constituidas por los terminales de reloj y de reset.
2. **Terminales de direcciones y datos:** incluye 32 líneas para datos y direcciones multiplexadas en el tiempo. Las otras líneas del grupo se utilizan para interpretar y validar las líneas de señal correspondientes a los datos y a las direcciones.
3. **Terminales de control de la interfaz:** controlan la temporización de las transferencias y proporcionan coordinación entre los que las inician y los desti-

natarios.

4. **Terminales de arbitraje:** a diferencia de las otras líneas de señal del PCI, éstas no son líneas compartidas. Cada maestro del PCI tiene su par propio de líneas que lo conectan directamente al árbitro del bus PCI.
5. **Terminales para señales de error:** indican errores de paridad, u otros.
6. **Terminales de Interrupción:** necesarias para los dispositivos que deben generar peticiones de servicio. Igual que los terminales de arbitraje, no son líneas compartidas si no que cada dispositivo PCI tiene su propia línea o líneas de petición de interrupción a un controlador de interrupciones.
7. **Terminales de Soporte de Cache:** necesarias para permitir memorias cache en el bus PCI asociadas a un procesador o a otro dispositivo. Estos terminales permiten el uso de protocolos de coherencia de cache de sondeo de bus.
8. **Terminales de Ampliación a Bus de 64 bits:** incluye 32 líneas multiplexadas en el tiempo para direcciones y datos y se combinan con las líneas obligatorias de dirección y datos para constituir un bus de direcciones y datos de 64 bits. Hay otras líneas de este grupo que se utilizan para interpretar y validar las líneas de datos y direcciones.
9. Por último, hay dos líneas que permiten que dos dispositivos PCI se pongan de acuerdo para usar los 64 bits. Las terminales de Test (JTAG/Boundary Scan) señales que se ajustan al estándar IEEE 1149.1 para la definición de procedimientos de test.

Una descripción de las terminales del bus PCI se describen en los cuadros B.1 y B.1. En donde los símbolos que se presentan tienen el siguiente significado:

- * Solo para un bus de 64 bits
- # Activa en nivel bajo
- S/T/S Sostenida en tercer estado
- I Entrada
- O Salida
- T/S Bidireccional (tercer estado)
- Op/D Señal con Drenador abierto

Nombre	**Tipo**	**Descripción**
CLK	I	Reloj del bus. Señal de sincronización de todas las señales, excepto la de inicialización de todos los registros Rsti.
RST#	I	Reset asíncrono. Pone en un valor conocido a todos los registros y máquinas de estado del sistema.
AD[31:0]	T/S	Datos/Direcciones multiplexados. Las direcciones coinciden con la validación de la señal *Framei* y los datos con la señal de escritura *Irdyi* y la de lectura *Trdyi*
C/BE#[3:0]	T/S	Comando/Byte activo. En la fase de direccionamiento recogen el comando asociado al tipo de transferencia. En fase de datos su significado se relaciona con la validación de los bytes de datos.
PAR	T/S	Paridad. Bit de control de paridad impar. Puesto por el maestro en las transacciones de escritura y por el esclavo en las de lectura.
FRAME#	S/T/S	Transacción activa. Gobernada por el maestro del bus para indicar inicio y duración de una transacción.
TRDY#	S/T/S	Esclavo listo. Puesta por el esclavo para indicar que está listo. En lectura indica que el bus PCI tiene datos válidos, en escritura que está listo para recibir datos.
IRDY#	S/T/S	Entrada lista. Gobierna la transacción sobre el esclavo junto con la señal *Trdyi*. Una fase de dato sólo tiene lugar cuando ambas señales están validadas. Su activación durante una operación de escritura indica que el bus PCI tiene datos válidos, en lectura que está preparado para recibir datos.
STOP#	S/T/S	Solicitud de parada de transacción. Puesta por el esclavo para solicitar al maestro el final de la transacción en curso.
LOCK#	S/T/S	*Operación atómica, por lo tanto necesita múltiples transacciones para ser completada.
IDSEL	I	Dispositivo seleccionado. Habilitación de dispositivo durante las transferencias de configuración exclusivamente.
DEVSEL#	S/T/S	Selector de dispositivo. Informa al maestro que el esclavo direccionado reconoce la dirección.

Cuadro B.1: *Descripción de las terminales utilizadas por el bus PCI.*

continuación de la tabla de descripción de las terminales		
Nombre	**Tipo**	**Descripción**
REQ#	T/S	Petición del bus. Indica al árbitro que el dispositivo correspondiente solicita utilizar el bus. Es una línea punto a punto específica para cada dispositivo.
GNT#	T/S	Concesión del bus. Indica al dispositivo que el árbitro le ha cedido el acceso al bus. Es una línea punto a punto específica para cada dispositivo.
PERR#	S/T/S	Error de paridad. Señal activada por el agente que recibe los datos dos ciclos después de que se detecte el error de paridad.
SERR#	O/D	Error del sistema. Reporta errores del sistema provocado por error de paridad u otra causa que deteriore la comunicación.
SBO#	I o O	*Snoop BackOff.
SDONE	I o O	*Snoop DONE.
AD[63:32]	T/S	*Bus de dirrección/datos para 64 bits.
C/BE#[7:4]	T/S	*Comando/Byte activo.
REQ64#	S/T/S	*Petición del bus.
ACK64#	S/T/S	*Solicitud de tranfeencia usando un bus de 64 bits.
PAR64	T/S	*Paridad par para *AD[63:32]* y *C/BE#[7:4]*.
TCK	I	*Prueba de reloj.
TDI	I	*Prueba del dato de entrada.
TDO	O	*Prueba del dato de entrada.
TMS	I	*Prueba del selector de modo.
TRST#	I	*Prueba del reset.
INTA#, INTD#	Op/D	Líneas de interrupciones. Indica la existencia de una interrupción a ser resuelta por el maestro.

Cuadro B.2: *Descripción de las terminales utilizadas por el bus PCI (continuación).*

B.3. Espacio de configuración

El estándar PCI 2,2 exige que todos los dispositivos PCI deban implementar 256 Bytes o 64 *DWORDS* de espacio de configuración. La finalidad es de albergar información relativa a la identificación del dispositivo, habilitación de funcionalidades, reconocimiento del espacio de direcciones y dirección base de los registros. Este espacio se compone de una región predefinida o cabecera de 64 Bytes y otra región dependiente del dispositivo de 192 Bytes, parte de la cual es obligatoria. En la figura B.2 se muestra un esquema de la cabecera del espacio de configuración con los registros obligatorios y optativos del la interfaz PCI.

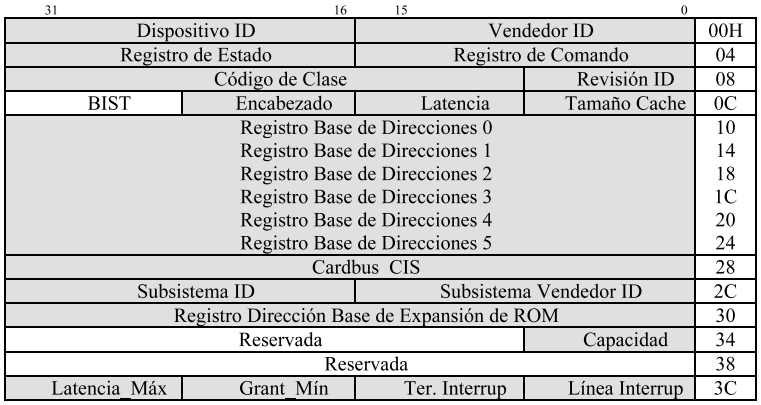


Figura B.2: *Espacio de configuración con los registros obligatorios y optativos del la interfaz PCI.*

La funcionalidad de un dispositivo PCI está descrita por el registro de estado. Este registro está asociado a una serie de operaciones (comandos) que puede realizar la interfaz. Un ordenador al arrancar realiza ciclos de configuración en todos los dispositivos conectados al bus PCI. Parte de la información presente en el espacio de configuración de cada dispositivo PCI permite a la BIOS del sistema habilitar el dispositivo PCI. También reservar un espacio de memoria en el ordenador o en la tabla de memoria del sistema, elaborada por el software de arranque. Los registros de dirección base (BAR's), son los encargados de informar a la BIOS del número y tamaño de los espacios de memoria o de entrada / salida necesarios, y de identificar una zona de memoria del sistema mapeada sobre un determinado dispositivo PCI. Este proceso se realiza en dos fases:

1. **En la primer fase**, el contenido de cada BAR es leído para determinar si se relaciona con un espacio de memoria o de entrada / salida (E/S) y qué tamaño precisa. El bit 0 de cada BAR indica si se trata de memoria o de E/S. Si es un 0 es de memoria y si es un 1 será un espacio de E/S. El peso binario del primer bit no nulo indica su tamaño según se trate de memoria o E/S.
2. **En la segunda fase**, el sistema decodifica esta información y escribe en cada BAR la dirección de memoria que le ha asignado el software de arranque. Además de todo esto, el número de dispositivos que pueden conectarse al bus PCI es de 256. La norma PCI 2,2 admite la jerarquización de buses incrementándose el número de dispositivos que pueden conectarse. El software de configuración debe ser capaz de realizar transacciones de configuración en todos los dispositivos PCI que residan más allá del puente PCI/host (bridge).

B.4. Descripción funcional

Cualquier transacción de datos llevada a cabo en el bus PCI tiene dos participantes: un iniciador (initiator) y un destinatario (target), también llamados *Master* y *Slave* respectivamente. El iniciador es el maestro del bus, es decir, el dispositivo que tiene el control del bus. El destinatario o esclavo es el dispositivo con el cual quiere comunicarse el maestro a través del bus.

La estrategia de control del bus PCI está basada en la asignación de acceso a través de un protocolo de simple petición/concesión, de forma que cualquier maestro que quiera hacer uso del mismo debe negociarlo con el resto de los maestros. Para ello se debe emplear un esquema de arbitraje centralizado. Por otro lado cada dispositivo con capacidad de ser maestro disponen de señales de petición y concesión del bus. Las señales que se conectan con el árbitro del bus PCI son; una línea de salida *REQ* de petición del bus y una línea de entrada *GNT* de concesión del bus.

Cuando un dispositivo conectado al bus desea llevar a cabo una transacción, siempre y cuando tenga la capacidad de ser maestro, activa la línea *REQ*. Cuando el árbitro concede el bus al dispositivo, el árbitro activa la línea *GNT* del dispositivo y este último puede usar el bus siempre y cuando este libre el bus o cuando termine la transacción en curso.

Los dispositivos conectados al bus están sincronizados con el flanco ascendente de la señal *CLK*. Por lo tanto, leen sus entradas en cada flanco ascendente generando sus salidas un poco después del flanco.

Una vez que el maestro del bus adquiere el control del mismo se define el tipo de transacción a realizar, mediante la señal *c/ben[3..0]* (*command/byte enable*), y con que dispositivo se va a efectuar la operación. Para saber que dispositivo es el indicado se utiliza la señal *ad[31..0]* (para un bus de direcciones/datos de 32 bits, que es el utilizado en este trabajo). A esta fase se le denomina fase de direccionamiento. El tipo de transacción será definida por la señal *c/ben[3..0]*, que es el comando que especifica la operación a realizar en función de la tabla B.3.

c/ben[3:0]	**Función**
0000	Reconocimiento de interrupción
0001	Ciclo especial
0010	Ciclo de lectura de I/O
0011	Ciclo de escritura de I/O
0100	Reservada
0101	Reservada
0110	Ciclo de lectura de memoria
0111	Ciclo de escritura de memoria
1000	Reservada
1001	Reservada
1010	Lectura de configuración
1011	Escritura de configuración
1100	Acceso múltiple a memoria
1101	Ciclo de direccionamiento dual
1110	Acceso a una línea de memoria
1111	Escritura de memoria con invalidación

Cuadro B.3: *Tabla del comando* c/ben[3:0] *y su respectiva función.*

Una vez decodificado el comando *c/ben[3..0]* se realiza la operación especificada. Así se inicia la transferencia de los datos. A esta etapa se le denomina fase de datos. Una transferencia típica se compone de una fase de direccionamiento y de una o más fases de datos permitiéndose realizar una transferencia a ráfagas.

B.4.1. Descripción funcional del MegaCore pci_mt32.

Aquí se realiza la descripción funcional del MegaCore **pci_mt32**, creada con el MegaWizard Plug-In Manger el cual es ejecutado en el software Quartus II de

Altera. Para identificar la función creada e identificar con que componentes se comunica se muestra la figura B.3. La interfaz PCI se localiza entre el Bus PCI y específicamente con los bloques `backend` y `ddr_cntrl_top`, localizados dentro del diseño implementado (`stratix_top`), llamándosele lado local.

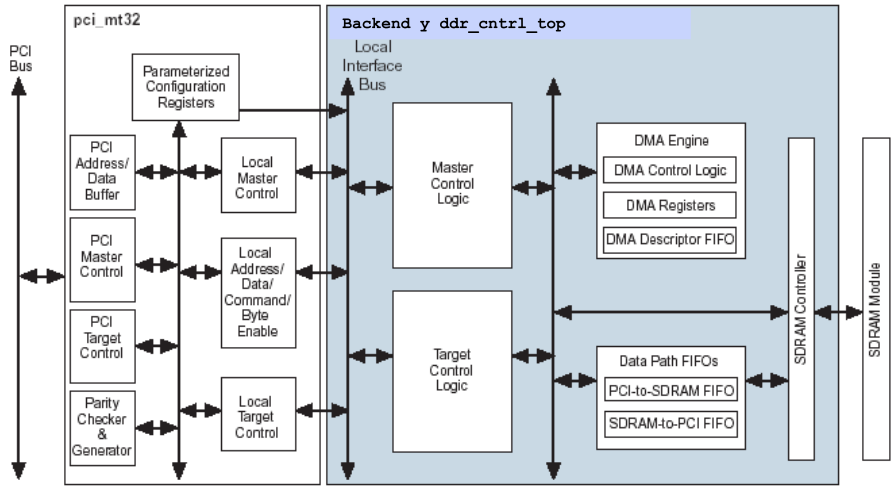


Figura B.3: *Diagrama a bloques en donde se identifica el bus PCI, la función* ***pci_mt32*** *y una parte del proyecto* `stratix_top`.

La función creada, en este trabajo, es una interfaz de 32 bits. Un diagrama a bloques de la interfaz, con las señales que la conforman, se muestra en la figura B.4.

Debido a que las señales del bus PCI se utilizan dentro del lado local es necesario identificarlas unas de otras. Para identificar las señales fácilmente se utilizan letras, delante de cada señal, para mostrar a que bloque se conecta dicha señal. Por ejemplo: la señal *lt_rdyn* indica que se conecta entre la interfaz PCI y el lado local, específicamente con el bloque de control Local Esclavo o Target. La señal *lm_rdyn* indica que se conecta también entre la interfaz PCI y el lado local, pero esta vez con el bloque de control Local Maestro o Master y la señal *trdyn*indica, por la letra "t" y la ausencia del guión bajo (_), que se conecta entre el BUS PCI y la interfaz PCI y que se localiza en el bloque de control PCI Target.

Como se dijo anteriormente, una transferencia típica se compone de una fase de direccionamiento y una o más fases de datos. Entonces, para una transacción Target, en la fase de direccionamiento la función **pci_mt32** compara la dirección del bus PCI con los rangos de direcciones de los registros BAR 0 y BAR 1. Si la dirección decodificada coincide con el espacio de direcciones reservado para los registros BAR 0 y BAR 1, la función **pci_mt32** confirma con la señal *devseln* al bus PCI para aceptar

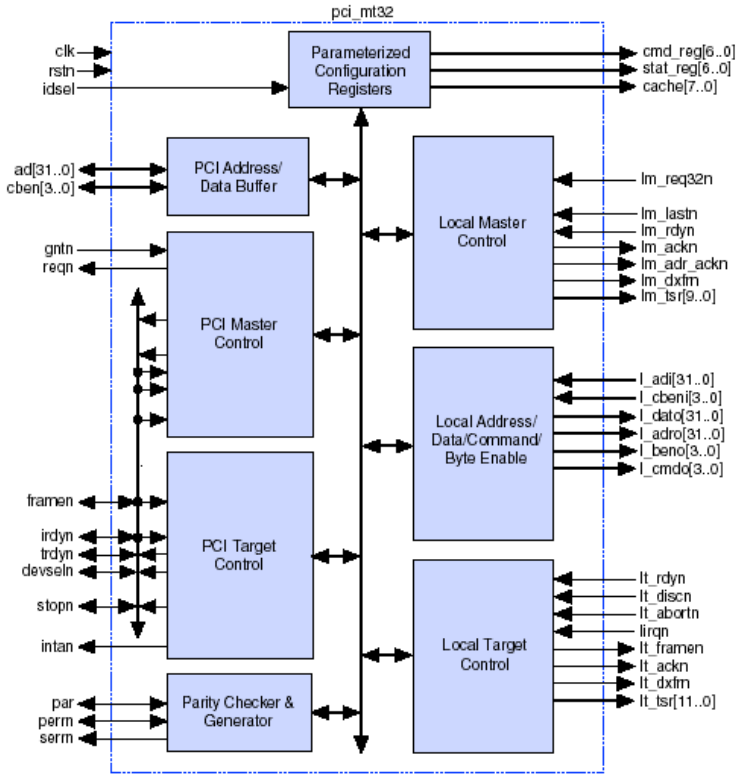


Figura B.4: *Diagrama a bloques de la función* ***pci_mt32****, creada por el MegaWizard Plug-In Manager, mostrando las señales que la conforman.*

la transacción. Después de esto la función **pci_mt32** activa las señales *lt_framen* y *lt_tsr[11..0]*, dentro del módulo `backend`, la lógica de control Target notificando que ha sido solicitada una transacción Target. La lógica de control Target (`targ_cntrl`) utiliza las señales *l_adro* y *lt_tsr[1..0]*, de la función **pci_mt32**, para determinar si la transacción es para la DDR SDRAM, los registros DMA o los buffer FIFO DMA. También es decodifica la señal *l_cmdo[3..0]* para determinar si es escritura o lectura a memoria o a I/O o de configuración. Para la función PCI aquí diseñada los BAR's sólo reservan espacio de memoria.

Posteriormente el bloque denominado CAD (command/address/data) localizado en la lógica de control Target, dentro del módulo `backend`, realiza la trasferencia de datos a través de una memoria FIFO utilizada, llamada PCI-to-SDRAM o SDRAM-to-PCI, según sea para escritura o lectura. Un diagrama a bloques que muestra la estructura a detalle de la lógica de control Target es mostrado en la figura B.5.

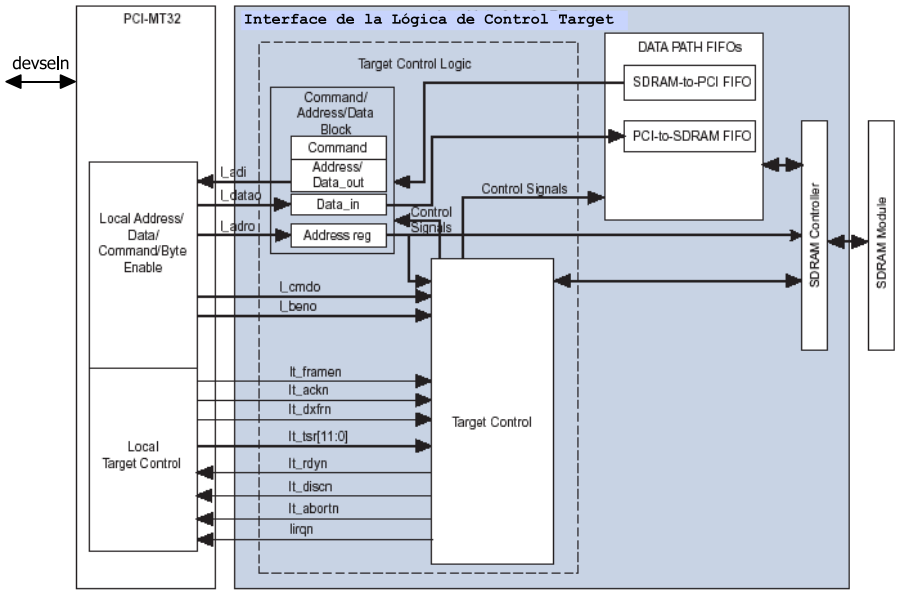


Figura B.5: *Diagrama de los bloques y señales que intervienen en una transferencia PCI-Target, en donde se identifica la lógica de control target.*

Lectura de memoria PCI a Target.

Si es decodificada la señal *l_cmdo* y se conoce que se solicitó una operación de lectura entonces la lógica de control Target activa la señal *lt_discn*. Esto es debido a que requiere un tiempo mayor a 16 ciclos de reloj para proporcionar los primeros datos a la FIFO SDRAM-to-PCI. Después de esto, la lógica de control Target activa la señal *lt_rdyn* para aceptar un ciclo de lectura de memoria y entonces los datos de la memoria FIFO SDRAM-to-PCI son transferidos al bus PCI hasta que la señal *lt_framen* es desactivada por la función pci_mt32.

Por otro lado si el número de palabras, dentro de la FIFO SDRAM-to-PCI, cae por debajo de un umbral establecido se activa la señal *lt_discn* para terminar la lectura ya que la memoria DDR SDRAM no puede proporcionar una lectura los datos en ese momento. Posteriormente el Master deberá solicitar nuevamente la lectura del resto de los datos. Al concluirse la trasferencia de lectura de los datos se desactiva la señal *lt_framen*.

Escritura de memoria PCI a Target.

Si es decodificada la señal *l_cmdo* y se conoce que se solicitó una operación de escritura, entonces la lógica de control Target escribe los datos en la memoria FIFO PCI-to-SDRAM FIFO hasta que la señal *lt_framen* es desactivada por la función **pci_mt32**. La interfaz de la SDRAM lee los datos de la memoria FIFO PCI-to-SDRAM y los escribe en la memoria DDR SDRAM hasta que la FIFO PCI-to-SDRAM se encuentre vacía concluyendo con la transacción.

En caso de que la FIFO PCI-to-SDRAM llegue a llenarse entonces la lógica de control Target activa la señal *lt_discn* deteniendo la escritura en la FIFO PCI-to-SDRAM. Posteriormente el Master deberá reiniciar el ciclo de escritura en la SDRAM para escribir el resto de los datos. El proceso concluye cuando la interfaz de la SDRAM lee los datos de la memoria FIFO PCI-to-SDRAM y los escribe en la memoria DDR SDRAM.