ESCUELA SUPERIOR DE INGENIEROS
UNIVERSIDAD DE SEVILLA

Ingeniería Superior de Telecomunicación

Proyecto Fin de Carrera

Implementación sobre plataforma reconfigurable de un sistema de visión en un único chip con hardware específico para el procesamiento de imágenes: evaluación de prestaciones y limitaciones

Autor del proyecto:
José Fernández Pérez

Tutor del proyecto:
Ricardo Carmona Galán

Ponente del proyecto:
Óscar Guerra Vinuesa

Departamento de Electrónica y Electromagnetismo
Universidad de Sevilla
Instituto de Microelectrónica de Sevilla-CNM-CSIC

Sevilla, 19 de Mayo de 2006

Agradecimientos:

Este proyecto ha sido realizado gracias a Ricardo Carmona, por su dirección, atención y consejos que han resultado claves en el resultado final. Gracias a mis compañeros en el IMSE-CNM, en especial a Luis y Carlos, por su disposición y apoyo. Mención especial para Francisco Javier Sánchez, sin cuya ayuda no hubiera sido posible. Gracias a Óscar Guerra y Santiago Sánchez por su colaboración. Gracias a mi familia y amigos por apoyarme incondicionalmente y a Rocío por estar a mi lado.

ÍNDICE

1. INTRODUCCIÓN

La visión es para los humanos, como primates superiores, la modalidad sensorial dominante, esto es, la vía principal para la adquisición de información procedente de nuestro entorno. A pesar de que los avances tecnológicos han permitido la difusión casi universal de dispositivos de bajo coste para la captura de imágenes, la visión es una de las capacidades sensoras menos explotada hasta el momento en lo relativo a sistemas autónomos de sensado-procesamiento-actuación en campos como la robótica, la automoción, la inteligencia ambiental, la biomedicina, etc. Los motivos principales están en la dificultad de manejar el flujo masivo de datos contenido inicialmente en el estímulo visual de manera eficiente. La típica arquitectura de Von Neumann, propia de los computadores de nuestros días, resulta ineficiente desde el punto de vista energético o bien, dicho de otro modo, simplemente resulta incapaz de realizar determinadas tareas en tiempo real ateniéndonos a unas dimensiones prácticas del sistema. Frente a estos sistemas convencionales basados en la captura más digitalización más procesamiento serializado, la Naturaleza ha sido capaz de desarrollar estructuras sensoras muy específicas. En los órganos sensoriales naturales, las tareas más fatigosas, con una mayor demanda computacional, son asignadas a conglomerados de dispositivos de cómputo elementales relativamente simples, lentos e imprecisos. A pesar de todo esto son capaces de realizar sus tareas consumiendo bastante menos energía que las alternativas artificiales. Sin mencionar el nivel de integración de los órganos sensores en el esquema de funcionamiento del sistema nervioso central, en el que la interpretación de los estímulos provenientes del entorno requiere de la interacción entre percepción y memoria.

Una de las herramientas empleadas por la evolución para llegar a estas soluciones optimizadas es el uso extensivo de la adaptación. Por un lado, los dispositivos biológicos destinados al procesamiento de información de tipo sensorial suelen poseer una arquitectura adaptada a la naturaleza del estímulo, o sea, están basados en redes de procesadores masivamente paralelas. Por otro lado, la adaptación de ciertos parámetros del sistema puede emplearse, mediante entrenamiento y/o aprendizaje, para corregir disparidades y atenuar los errores, permitiendo niveles de precisión aceptables mediante circuitería relativamente imprecisa. De modo que, como primer paso en el desarrollo de procesadores eficientes en el tratamiento de información de carácter sensorial, podemos pensar en una adaptación de la arquitectura a la naturaleza del estímulo.

En este proyecto nos planteamos la realización de un procesador de imágenes especializado que libere al microprocesador de propósito general de las tareas para las que no resulta especialmente eficiente. Este coprocesador visual, va a constar de una serie de elementos básicos que realicen las operaciones descritas en los algoritmos de visión de manera eficiente. La adaptación del sistema a la naturaleza del estímulo va a estar basada en la replicación apropiada de estas unidades básicas de procesamiento en función del análisis de las especificaciones del problema. Los objetivos del proyecto van a ser, por un lado, el estudio de la arquitectura del sistema que permita plantear la implementación de aplicaciones completas de visión artificial en un único chip, además de la cámara, y en tiempo real. Por otro, constatar que este sistema puede realizarse,

con las limitaciones que nos encargaremos de reseñar, sobre una plataforma hardware reconfigurable, como son las FPGA's. Finalmente, el prototipo diseñado nos permitirá establecer las prestaciones y limitaciones de un sistema de visión basado en esta arquitectura, y hacer previsiones sobre su posible dimensionado y escalado para la aplicación a problemas reales y, en caso de que el mercado para dichas aplicaciones lo permitiera, su posible implementación como un ASIC.

1.1 Análisis del estado del arte

Como hemos dicho, el procesamiento de imágenes en tiempo real requiere una elevada capacidad de cómputo. Supongamos, por ejemplo, un flujo de 25fps[1] en formato QCIF [1], lo que significa un caudal de datos de 0.63Mpixels/s. Si para una aplicación en particular necesitáramos realizar 20 convoluciones de cada imagen con máscaras espaciales de 3x3 (o sea que por cada píxel tuviéramos que realizar 9 productos y 8 sumas), tendríamos que ser capaces de desarrollar una capacidad de cómputo de unos 214MOPS[2].

Los sistemas basados en un único procesador de propósito general pueden no resultar adecuados para generar una respuesta precisa en escalas de tiempos demasiado estrechas. En aplicaciones como, por ejemplo, la detección de obstáculos o el seguimiento de personas y objetos en un entorno cambiante, el tiempo de respuesta del sistema debe ser del orden de los milisegundos. Por un lado tenemos que el procesamiento serializado que realizan los procesadores de propósito general con arquitecturas convencionales genera cuellos de botella que limitan la tasa de procesamiento del sistema, pudiendo hacer inviable el procesamiento de imágenes en tiempo real. Por otro lado, además de encargarse del procesamiento de las imágenes, el microprocesador tiene que dedicar tiempo a otras cuestiones laterales, o centrales, como la comunicación con los periféricos y la ejecución de un programa global. Si además contamos con restricciones en el consumo de potencia impuestas por las aplicaciones (robots autónomos, cámaras inteligentes y portátiles), se hace muy complicado alcanzar las prestaciones de potencia de cálculo requeridas mediante un procesador de propósito general. Por ejemplo, el Intel Pentium 4EE [2], un procesador de los más recientes, es capaz de desarrollar 9730MIPS[3], pero lo hace a costa de una alta frecuencia de reloj (3.2GHz) y un elevado consumo de potencia (103W). Esto equivaldría a una capacidad de cómputo por potencia consumida de 0.094MIPS/mW. Otros ejemplos serían el AMD Athlon 64 FX (Dual Core) [3], cuya capacidad de cómputo por milivatio llega a los 0.201MIPS/mW, y el Hitachi SH7705 [4], diseñado específicamente para aplicaciones portátiles, con un throughput de 0.865MIPS/mW. Según nuestros cálculos, basados en los resultados obtenidos de la implementación de un sistema de visión on-chip en una FPGA (Virtex II Pro de Xilinx), sería posible desarrollar estructuras especializadas para el procesamiento de imágenes en paralelo, o al menos el procesamiento de parte de la imagen en paralelo, que fueran capaces de poner en

[1] *Frames per second:* imágenes por segundo
[2] MOPS: Millones de operaciones (sumas, multiplicaciones, etc.) por segundo.
[3] MIPS: Millones de instrucciones por segundo. Esta medida no es muy exacta dado que una instrucción puede precisar de un número variable de ciclos de reloj y una operación aritmética puede necesitar de varias instrucciones para ser realizada.

juego una capacidad de cómputo por encima de los 2MOPS/mW (como veremos al final de esta memoria). Y esto en una plataforma reconfigurable, no especialmente diseñada al efecto. Si estas estructuras se llevaran a un ASIC en el que eliminámaramos el *overhead* de *hardware* que las FPGA's contienen para su programación y reconfiguración, comprobaríamos que la adaptación de la arquitectura del sistema a la propia naturaleza de la señal a procesar permite afrontar de manera eficiente, desde el punto de vista energético, la carga computacional de las aplicaciones más demandantes.

1.2 Implementación en plataforma reconfigurable

El *hardware* reconfigurable, como las FPGA's (*Field Programmable Gate Arrays*) o los PLD's (*Programmable Logic Devices*), está comenzando a resultar cada vez más atractivo para problemas de procesamiento digital de señales, incluyendo tareas de procesamiento de imágenes y de visión artificial. Esto se debe principalmente a que cada vez existen dispositivos más potentes, esto es, con mayor número de recursos, tanto de lógica de grano fino como de lógica de grano grueso, a precios muy convenientes, y a que el desarrollo de sistemas basados en estos dispositivos resulta muy rentable desde el punto de vista del tiempo y de los costes de desarrollo.

Frente a dispositivos programables con una arquitectura cerrada, como pueden ser los microprocesadores de propósito general o los DSP's, las FPGA's poseen flexibilidad para explotar la naturaleza inherentemente paralela de muchos problemas de visión. Por ejemplo, algunos algoritmos requieren la aplicación del mismo operador de manera local a cada píxel, como ocurre con las máscaras de convolución. En un procesador serializado esta operación podría llevar bastante tiempo, pero en un FPGA puede configurarse un sistema en el que tengan lugar simultáneamente múltiples convoluciones. Es cierto, no obstante, que los dispositivos programables de arquitectura cerrada suelen estar acompañados de librerías de funciones y entornos de programación amigables que facilitan el desarrollo de aplicaciones. Sin embargo, la posibilidad de moldear la estructura interna de los sistemas basados en FPGA's los hace especialmente adecuados para el desarrollo de aplicaciones que van a depender mucho de la adaptación de la arquitectura al problema, lo que es típico cuando se trabaja con información masiva de carácter sensorial.

Frente a dispositivos específicos, ASIC's (*Application Specific Integrated Circuit*), que poseen una programabilidad muy restringida, el diseño en plataformas reconfigurables supone un menor riesgo en el desarrollo del sistema. Si el diseño final implementado posee algún fallo, siempre se puede reprogramar el dispositivo, lo que se conoce informalmente como *painless design*. Además, el ciclo *diseño-implementación-test-depurado* puede llevarse a cabo en unas horas, y no tardar del orden de meses, como ocurre en los ASIC's, lo que hace que realizar pequeños cambios en el diseño resulte una tarea sencilla.

También, el time-to-market de un sistema basado en FPGA's es bastante menor que el de un ASIC, ya que cualquier cambio, innovación o mejora puede introducirse mediante la programación en la herramienta de síntesis y la reconfiguración del dispositivo, evitándose la elaboración de nuevas máscaras litográficas, que implicarían un mayor tiempo de fabricación. Aparte de que, para tiradas no demasiado elevadas el coste de fabricación de un ASIC puede

resultar inviable desde el punto de vista comercial, ya que el desarrollo de las máscaras que se usarán para fabricar dispositivos posee un coste económico bastante elevado.

No todo son ventajas en el uso de las FPGA's. En el campo del procesamiento digital de señales, el empleo de aritmética de punto flotante resulta en una importante demanda de recursos internos para la FPGA. En nuestro caso, esto redundaría en un menor grado de paralelización de las operaciones por falta de recursos computacionales, y por tanto una menor velocidad de procesamiento del sistema. Además, comparadas con los ASIC's, las implementaciones que usan FPGA's son típicamente menos eficientes debido al gasto de recursos del dispositivo que son destinados a la circuitería de configuración, incluyendo la E/S y las celdas SRAM requeridas para almacenar el diseño actual. Esto lleva a dispositivos y con un mayor perfil físico y un mayor consumo de potencia.

Vamos a implementar nuestro sistema de visión en una FPGA, dado que, en este punto, estamos interesados en evaluar las posibilidades de un sistema de visión para diferentes aplicaciones en un único chip. Se trata de un proyecto de investigación, es decir, no existen unas especificaciones cerradas, como sería el caso en una aplicación con fines comerciales. Además, en nuestro caso, no vamos a tratar con algoritmos que utilicen aritmética de punto flotante, por lo que evitamos los inconvenientes principales que poseían las FPGA's al respecto.

1.3 Aplicaciones

El ámbito de aplicación del procesamiento digital de imágenes es bastante amplio, apareciendo en muchas áreas científicas, técnicas e industriales. Por ejemplo, en *aplicaciones industriales,* la visión artificial se emplea en el control de los procesos de producción, la verificación de la calidad de los productos y procesos [5], el guiado y posicionamiento de robots y máquinas [6]. En *automoción,* el tratamiento de imágenes permite llevar a cabo la detección de obstáculos, señales y otros elementos relacionados con la conducción, y por tanto formar parte de sistemas de mantenimiento de trayectoria y actitud, alarmas de colisión y corrección de la trayectoria, y otras ayudas a la detección de posibles peligros por parte del conductor [7]. En el desarrollo de robots autónomos, la navegación, planificación de los desplazamientos, ubicación en el entorno, y reconocimiento de objetos o personas, la visión juega un papel muy importante [8]. En el campo de la *medicina* y la *biología* se usa el procesamiento de imagen en la segmentación de resonancias magnéticas del cerebro [9], en la caracterización de secuencias temporales de imágenes del corazón [10], en la segmentación de imágenes de biopsias de la piel [11], en la detección de microcalcificaciones en mamografías realizadas con de rayos X de gran resolución [12], en la segmentación de células endoteliales de la córnea [13], en la detección automática de células cancerosas [14], etc., todas ellas requiriendo una capacidad de cómputo que imposibilita a menudo el desarrollo de equipamientos portátiles para actuaciones de emergencia. En *aplicaciones geoespaciales* es muy importante el reconocimiento de formas en imágenes satelitales de la superficie terrestre, por ejemplo, para caracterizar formaciones nubosas en satélites con fines meteorológicos, o para identificación de accidentes y fenómenos geográficos [15]. En materia de *identificación* y *control de seguridad,* la extracción de características morfológicas de una imagen hace

posible el reconocimiento facial [16], la detección de objetivos en imágenes borrosas o subacuáticas [17], aplicaciones de control de tráfico [18], etc. En el *procesamiento de documentos* posibilita la extracción de información de caracteres impresos o manuscritos [19], así como el reconocimiento de símbolos musicales en partituras [20]. También resulta importante el empleo del procesamiento de imágenes en el desarrollo de la infraestructura necesaria para la implementación de la *inteligencia ambiental* [21]. En ella, artefactos inteligentes intercambian información con el medio que les rodea, también mediante la visión artificial.

1.4 Propuesta y estructura de la memoria

Con objeto de estudiar las posibilidades de un sistema de visión en un único chip, que contuviera algún tipo de adaptación de su arquitectura de cómputo a la naturaleza del estímulo visual, vamos a plantearnos el desarrollo del mismo en una plataforma reconfigurable. Por un lado, dispondremos de facilidades para adecuar la estructura interna del sistema, a nivel de hardware, a nuestras necesidades. Por otro lado, tendremos todo un entorno de herramientas que nos van a facilitar el diseño, la simulación, la síntesis y el test del sistema a un coste razonable, sobre todo teniendo en cuenta que no nos estamos centrando en una aplicación muy concreta de la visión artificial. Finalmente, vamos a poder extraer conclusiones interesantes a partir de la evaluación de las prestaciones y limitaciones de este sistema prototipo.

En la siguiente sección, vamos a hacer un breve repaso de los operadores básicos que suelen utilizarse para la realización de las diferentes tareas envueltas en el procesamiento de imágenes a bajo nivel, esto es, en el mismo plano de la imagen y trabajando directamente con los valores de los píxeles. En la Sección 3ª, describiremos la arquitectura del sistema de visión on-chip que hemos diseñado, haciendo un repaso de los diferentes elementos que la componen, su estructura interna y modos de programación y funcionamiento. En la Sección 4ª explicaremos los detalles de la implementación del sistema descrito en una FPGA, en concreto una Virtex II Pro. A continuación describiremos algunos ejemplos de la operación del sistema, para pasar en la Sección 6ª a la evaluación de los resultados obtenidos con el fin de establecer las prestaciones y limitaciones del sistema y hacer las previsiones correspondientes para futuras implementaciones. Finalmente, se exponen las referencias bibliográficas que hemos consultado en el desarrollo de este trabajo, y, como apéndice, las piezas de código más relevantes que se han desarrollado en el proceso de diseño, síntesis y test de este sistema.

2. OPERACIONES BÁSICAS EN ALGORITMOS DE PROCESAMIENTO DE IMÁGENES

En este apartado vamos a tratar aspectos teóricos acerca del tratamiento de imágenes, que sirvan como introducción para entender la filosofía de diseño del sistema.

Para empezar, hay que especificar qué vamos a entender como imagen. Una imagen bidimensional monocromática es una función continua de las coordenadas de un punto en el plano focal (x, y) que es proporcional a la intensidad luminosa de dicho punto, que también es una variable continua. Las imágenes con las que trabajaremos serán imágenes resultantes de un proceso de digitalización, bidimensionales, que estarán discretizadas y que denotaremos como (i, j), cuyos píxeles tendrán unos valores de nivel de gris comprendidos entre 0 y 255, es decir, cada valor vendrá representado por 8 bits cuando sea implementado. De modo que de ahora en adelante, nos vamos a referir indistintamente a las imágenes digitalizadas o no. El contexto va a clarificar el tipo de imagen al que nos referimos. Habitualmente, trataremos con imágenes como funciones continuas en el espacio, el tiempo y la magnitud cuando analicemos matemáticamente los operadores necesarios para desarrollar las tareas de procesamiento. Las imágenes digitalizadas aparecerán cuando nos planteemos la implementación de estos operadores, bien en un procesador de propósito general o bien en un hardware específico de procesamiento digital.

El formato de representación es como sigue:

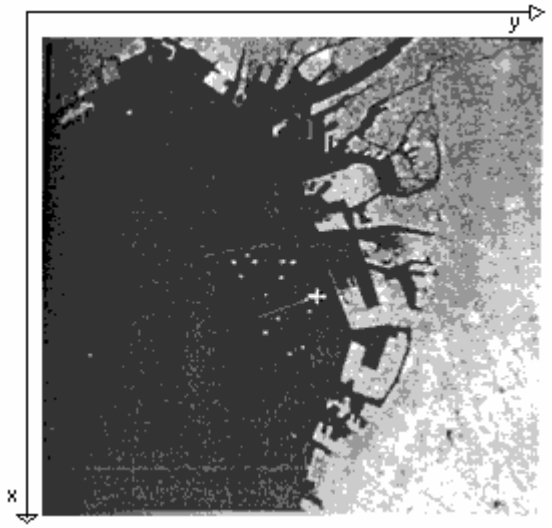


Figura 2.1. Convención de ejes usada para imágenes

Un primer objetivo del procesamiento es la generación de una imagen, resultante de tratar la imagen original digitalizada, de modo que cumpla ciertos requisitos según la necesidad que tengamos. Valga como ejemplo la obtención de una versión suavizada, filtrada o umbralizada de la imagen a procesar. De todos modos la salida del sistema no tiene por qué ser formalmente una imagen. De hecho, vamos a dotar al sistema de un procesador de propósito general para

que se pueda plantear la realización de tareas de procesamiento de más alto nivel de abstracción.

Vemos en qué consiste el procesado de imágenes y cuáles son las operaciones que tienen lugar en el tratamiento de las mismas.

2.1 Operaciones básicas

2.1.1 Operaciones aritméticas con dos imágenes

De aquí en adelante vamos a representar las imágenes monocromáticas digitalizadas como matrices de tamaño MxN, del tipo:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & \dots & a_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ a_{M1} & a_{M2} & \dots & \dots & a_{MN} \end{pmatrix}; \qquad (2.1)$$

en la que los elementos de la matriz cumplen:

$$a_{ij} \in \{0,1,2,..,255\},\ \forall i, j \in Z^+ \ / \ i \in [1, M],\quad j \in [1, N]; \qquad (2.2)$$

Las operaciones aritméticas básicas que aquí consideramos son la suma, resta y multiplicación por una constante. Pasamos a describir cada una de estas operaciones a continuación:

Suma de imágenes

Consideramos dos imágenes A y B, descritas de la forma indicada en las ecuaciones (2.1) y (2.2), es decir A = (a_{ij}) y B = (b_{ij}), con las variables *i* y *j* recorriendo la imagen de tamaño MxN.

Decimos que la imagen C = (c_{ij}), también descrita según (2.1) y (2.2), es la suma de A y B si se cumple:

$$c_{ij} = a_{ij} + b_{ij};\ \forall i, j \in Z^+ \ / \ i \in [1, M],\quad j \in [1, N]; \qquad (2.3)$$

Según (2.2), los elementos de la matriz suma C deben tener valores enteros comprendidos entre 0 y 255, por lo que la suma satura al valor 255.

Resta de imágenes

Consideramos dos imágenes D y E, descritas de la forma indicada en las ecuaciones (2.1) y (2.2), es decir D = (d_{ij}) y E = (e_{ij}), con las variables *i* y *j* recorriendo la imagen de tamaño MxN.

Decimos que la imagen F = (f_{ij}), también descrita según (2.1) y (2.2), es la resta de D y E si se cumple:

$$f_{ij} = d_{ij} - e_{ij};\ \ \forall i, j \in Z^{+}\ /\ i \in [1, M],\quad j \in [1, N]; \tag{2.4}$$

Según la ecuación (2.2), los elementos de la matriz diferencia F, poseen valores comprendidos entre 0 y 255, por lo que la resta satura al valor 0.

2.1.2 Operaciones con una imagen y un escalar

Multiplicación por un escalar

Consideramos una imagen G, descrita de la forma indicada en las ecuaciones (2.1) y (2.2), es decir G = (g_{ij}), con las variables *i* y *j* recorriendo la imagen de tamaño MxN, y un escalar λ definido como número entero positivo[4].

Decimos que la imagen H = (h_{ij}), también descrita según (2.1) y (2.2), es la versión escalada de G si cumple:

$$h_{ij} = \lambda \cdot g_{ij};\quad \forall i, j, \lambda \in Z^{+}\ /\ i \in [1, M],\quad j \in [1, N]; \tag{2.5}$$

Según la ecuación (2.2), los elementos de la matriz escalada H, poseen valores comprendidos entre 0 y 255, por lo que el escalado satura al valor 255.

Comparación con un escalar

Para operar con imágenes digitalizadas vamos a utilizar las operaciones lógicas básicas (AND, OR, NOT) sobre los bits que representan cada uno de los valores de los píxeles. Una combinación adecuada de estas operaciones lógicas nos va a permitir realizar, por ejemplo, el umbralizado. Esta operación consiste en comparar los valores de los elementos de una imagen K, definida según (2.1) y (2.2), con un valor umbral *T*, definido para valores enteros comprendidos entre 0 y 255. El resultado es una imagen binaria, L, que, en nuestro caso, tomará los valores 0, si el valor del elemento evaluado es menor que el umbral, o 255, si el

[4] Para determinadas operaciones es posible que el factor de escalado no sea un número entero, pudiendo ser un número fraccionario, en cuyo caso se adaptaría la expresión (2.5). De todos modos el conjunto de valores de los elementos de la imagen resultante ha de respetar las restricciones indicadas por (2.2).

valor del elemento es igual o mayor que el umbral. Lo expresamos de una manera más formal:

$$l_{ij} = \begin{cases} 255 & k_{ij} < T \\ 0 & k_{ij} \geq T \end{cases} \quad \forall i, j \in Z^+ \ / \ i \in [1, M], \quad j \in [1, N]; \tag{2.6}$$

2.1.3 Operaciones con una imagen y una máscara espacial

Existen distintos métodos para el tratamiento de imágenes digitalizadas. Puede trabajarse sobre una imagen en el dominio espacio-temporal, en el que se actúa directamente sobre los píxeles de la misma, o bien, en el dominio frecuencial, mediante la aplicación de alguna transformada. El tratamiento de las imágenes que vamos a usar aquí es el procesado por máscaras. Este es un método de dominio espacio-temporal.

Las funciones de procesado de imágenes en el dominio espacial pueden expresarse de la siguiente forma:

$$g(i, j) = \hat{R}[f(i, j)], \tag{2.7}$$

donde $f(i, j)$ es la imagen digitalizada a procesar y $g(i, j)$ la imagen procesada, siendo $\hat{R}$ un operador sobre *f*, definido sobre un vecindario del píxel (*i*, *j*), es decir, sobre los píxeles que rodean al que está siendo evaluado en ese instante.

La figura 2.2 muestra el proceso:

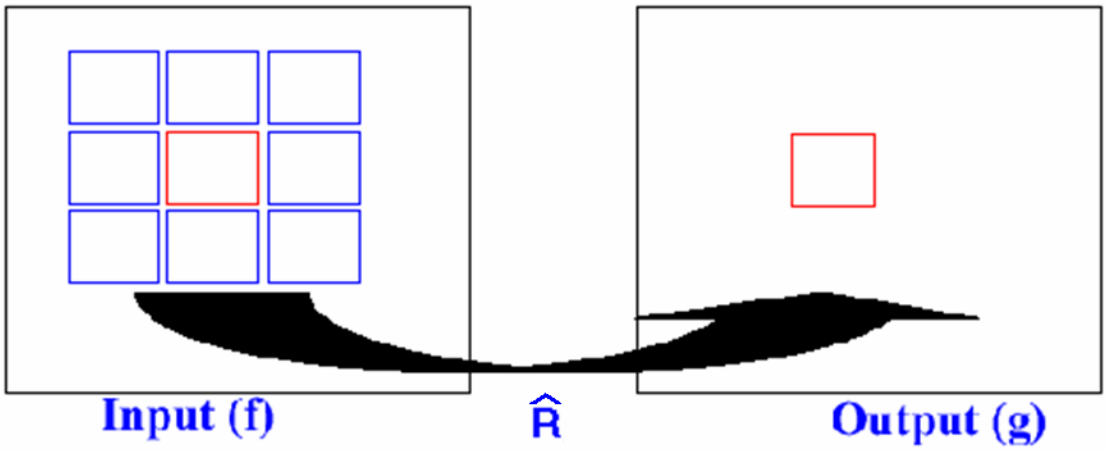


Figura 2.2. Procesado de una imagen usando una máscara 3x3

Se observa cómo se obtiene el valor de un píxel (*i*, *j*) en una imagen objetivo mediante el procesado del píxel (*i*, *j*) y de sus vecinos a través de una cierta función $\hat{R}$. La siguiente expresión muestra, usando notación matricial, que es la que usaremos de aquí en adelante, un ejemplo de máscara 3x3, cuyos coeficientes se muestran junto con sus posiciones relativas al píxel a procesar.

$$W_{ij} = \begin{pmatrix} w_{i-1,j-1} & w_{i-1,j} & w_{i-1,j+1} \\ w_{i,j-1} & w_{i,j} & w_{i,j+1} \\ w_{i+1,j-1} & w_{i+1,j} & w_{i+1,j+1} \end{pmatrix} \tag{2.8}$$

El procesado consiste en hacer que la máscara recorra toda la imagen píxel a píxel, generando la imagen objetivo de este modo. A este procedimiento lo llamamos convolución de una imagen. Consideramos que la imagen original viene dada por la matriz *F*, y que la imagen a obtener mediante la convolución de *F* con *W* se representa mediante la matriz *G*. Expresamos esto formalmente:

$$G = \left[W \otimes F \right], \tag{2.9}$$

Estando definidas *G* y *F* sobre las variables *i* y *j*, las cuales se encargan de recorrer la imagen, y *W* sobre *k* y *l*, cuyos valores están limitados al conjunto formado por {-1, 0, 1}. Desarrollamos la expresión anterior:

$$G(i, j) = [W \otimes F](i, j) = \sum_{k=-1}^{1} \sum_{l=-1}^{1} W(k,l) \cdot F(i+k, j+l) \tag{2.10}$$

2.2 Algunas funcionalidades básicas

2.2.1 Segmentación mediante la detección de discontinuidades

Dentro de la gran variedad de elementos que conforman el análisis de imágenes digitales, vamos a centrarnos en aquellos destinados a la segmentación de imágenes, es decir, realizar una subdivisión de la imagen en sus partes constituyentes u objetos.

Es la segmentación una herramienta básica para la comprensión o interpretación del contenido de una imagen, ya que constituye el primer paso para labores como el reconocimiento y descripción visual. Se basa en dos propiedades básicas de los valores del nivel de gris de los píxeles de una imagen: discontinuidad y similitud.

En la primera categoría, vamos a dividir en partes una imagen basándonos en cambios abruptos del nivel de gris de sus píxeles. Las principales áreas de interés dentro de esta categoría son:

- detección de puntos aislados,
- detección de líneas,
- detección de bordes[5] en una imagen.

Respecto a la segunda categoría, nos encontramos con técnicas basadas en:

- umbralizado,
- formación de regiones,
- unión y separación de regiones.

[5] Un borde es un conjunto de píxeles conectados que dividen dos regiones pertenecientes a diferentes objetos.

El concepto de segmentación de una imagen mediante el uso de estas técnicas basadas en discontinuidad y similitud de los valores de los niveles de gris de sus píxeles es aplicable tanto en imágenes estáticas como en dinámicas (variantes con el tiempo). En este apartado vamos a ver técnicas de detección de puntos, líneas y bordes en una imagen. Los métodos que aparecen suelen estar basados en pequeñas máscaras espaciales como la genérica 3x3 mostrada a continuación:

$$W = \begin{pmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{pmatrix}, \tag{2.11}$$

Al realizar la convolución:

$$R = w_1 z_1 + w_2 z_2 + \ldots + w_9 z_9 = \sum_{i=1}^{9} w_i z_i \ , \tag{2.12}$$

donde R es la repuesta de la máscara en cualquier punto de la imagen, w_i el valor del i-ésimo coeficiente de la máscara, y z_i el valor del nivel de gris del píxel i-ésimo dentro de la región de la imagen cubierta por la máscara.

La respuesta de la máscara está definida con respecto a su punto central, que es precisamente el píxel que está siendo procesado. Hay que tener en cuenta también las condiciones de contorno de la convolución aplicada sobre la imagen. En el caso de estar centrada la máscara en un punto del límite de una imagen, la respuesta será computada teniendo en cuenta sólo los vecinos del píxel que caigan dentro de la misma[6].

Detección de puntos

La detección de puntos en una imagen tiene aplicación en la eliminación de ruido y en el análisis de partículas. La máscara usada en la detección de puntos aislados en una imagen de fondo (*background*) constante es la mostrada a continuación:

$$D_p = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}, \tag{2.13}$$

Decimos que un punto aislado ha sido detectado en la posición en la cual está centrada la máscara si se cumple:

$$\left| R \right| > T \ , \tag{2.14}$$

donde T es un umbral no negativo y R es la respuesta de la máscara. Lo que estamos haciendo al realizar esta operación es medir las diferencias entre los niveles de gris del píxel procesado y de sus vecinos.

[6] Esto en una elección arbitraria que hemos tomado por conveniencia.

Detección de líneas

La detección de puntos en una imagen es una tarea directa, como hemos visto, pues basta con hacer pasar la máscara sobre la imagen y comparar con un umbral. Ahora subimos un peldaño de dificultad y encaramos el problema de la detección de líneas en una imagen. Veamos las siguientes máscaras:

$$D_{Lh} = \begin{pmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{pmatrix} \quad (2.15) \qquad D_{L45} = \begin{pmatrix} -1 & -1 & 2 \\ -1 & 2 & -1 \\ 2 & -1 & -1 \end{pmatrix} \quad (2.16)$$

$$D_{Lv} = \begin{pmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{pmatrix} \quad (2.17) \qquad D_{L-45} = \begin{pmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{pmatrix} \quad (2.18)$$

Nos fijamos en la máscara dada por la ecuación (2.15), si la convolucionamos con la imagen obtendríamos una respuesta de máscara mayor en las líneas (de un píxel de espesor) orientadas horizontalmente. Con *background* constante, la respuesta más alta la tendríamos precisamente cuando una línea horizontal pasara por la fila central de la máscara. Así, para cada máscara de las ecuaciones anteriores, obtendríamos detección de líneas de distinta orientación:

- Máscara (2.15): Detecta líneas orientadas horizontalmente.
- Máscara (2.16): Detecta líneas orientadas a 45º del eje horizontal.
- Máscara (2.17): Detecta líneas orientadas verticalmente.
- Máscara (2.18): Detecta líneas orientadas a –45º del eje horizontal.

Por tanto, supongamos que tenemos un punto y realizamos la convolución de las máscaras de las ecuaciones (2.15), (2.16), (2.17) y (2.18) en el vecindario 3x3 centrado en dicho punto. Si se cumple:

$$|R_i| > |R_j| \quad \forall j\,, \qquad (2.19)$$

siendo R_i y R_j las respuestas a las máscaras *i* y *j* respectivamente, tendríamos que ese punto es favorito para estar colocado en una línea de orientación dada por la máscara *i*.

Detección de bordes

Aunque la detección de puntos y líneas son elementos que forman parte de la segmentación, la detección de borde es, de lejos, el método más usado en la búsqueda de discontinuidades auténticas en los niveles de gris de una imagen. Esto es así porque la detección de puntos aislados y líneas delgadas no tienen apenas repercusión en la mayoría de casos prácticos.

Un borde es la frontera entre dos regiones con propiedades ligeramente distintas en sus niveles de gris. Asumimos que las regiones en cuestión son lo

suficientemente homogéneas como para que la transición entre dos regiones pueda ser determinada basándonos únicamente en las discontinuidades del nivel de gris de la imagen. Si no se cumple esta hipótesis no se usaría detección de borde, sino otras técnicas, como las orientadas a región y el umbralizado.

La idea que consideramos aquí es la aplicación de un operador derivativo local. Para ilustrar esto nos ayudaremos de la siguiente figura:

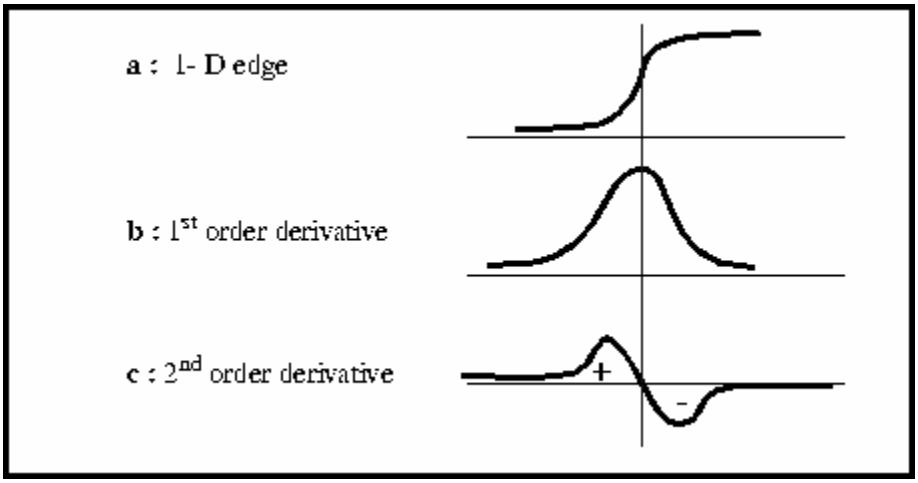


Figura 2.3. Comportamiento de operadores derivativos en una discontinuidad.

La figura 2.3 muestra el efecto que tienen los operadores derivativos de primer (**b**) y segundo orden (**c**) en una zona con discontinuidades. Ésta se muestra en (**a**).

La primera derivada de un borde modelado de esta manera es nula en los puntos donde el nivel de gris permanece constante y asume un valor más o menos constante durante la transición en el nivel de gris. Por otra parte, la segunda derivada es nula en todos los puntos, excepto en los comprendidos entre el comienzo y el final de la transición.

Basándonos en estos conceptos, y mirando la figura 2.3, podemos observar que la magnitud de la primera derivada puede ser usada para detectar la presencia del borde, de manera que si realizamos una aproximación de la primera derivada y la expresamos en forma matricial, como veremos más adelante, al convolucionar esa matriz con la imagen original obtendríamos valores más altos en las posiciones correspondientes a los bordes presentes a la imagen, permitiendo su diferenciación del resto de componentes de la misma. Del mismo modo, el signo de la segunda derivada puede usarse para indicar si un píxel del borde cae en la zona oscura o clara del mismo También hemos de observar cómo la segunda derivada se anula justamente en el centro de la transición, lo cual también es una propiedad aprovechable.

La discusión anterior hacía referencia al caso de una discontinuidad unidimensional horizontal. Para el caso de una imagen (2 dimensiones) utilizaremos un argumento similar. Para ello bastaría con definir un perfil perpendicular a la dirección de la discontinuidad e interpretar los resultados tal como hicimos en la discusión anterior. La primera derivada en cualquier punto de la imagen, y en una dirección determinada, está relacionada con el gradiente en ese punto ($\vec{\nabla} f$), mientras que la segunda derivada viene dada por el laplaciano ($\nabla^2 f$), como se detalla en las ecuaciones (2.20) y (2.41).

El método más común de diferenciación en aplicaciones de procesamiento de imagen es el gradiente. Supongamos que tenemos una imagen $f(x,y)$ definida de la forma:

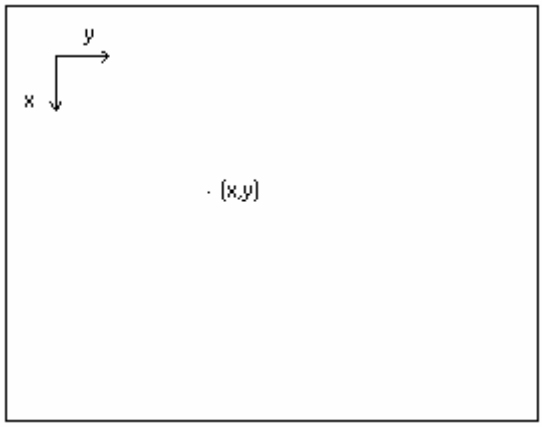


Figura 2.4. Representación de imagen y definición de sus ejes.

Si tomamos la imagen como una función en *x* e *y*, el gradiente de *f* vendría dado por el vector bidimensional:

$$\vec{\nabla} f = \frac{\partial f}{\partial x}\vec{u}_x + \frac{\partial f}{\partial y}\vec{u}_y \quad , \tag{2.20}$$

donde aparecen $\vec{u}_x$ y $\vec{u}_y$, que son los vectores unitarios de una base ortonormal que describe el plano (x, y). Del análisis vectorial sabemos que el gradiente es un vector que apunta hacia la dirección en la cual es mayor la variación de *f* en el punto (*x*, *y*). A nosotros nos interesa, para la detección de bordes, la magnitud de este vector, que será denotada por $\left|\vec{\nabla} f\right|$, donde:

$$\left|\vec{\nabla} f(x,y)\right| = \left[\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2\right]^{1/2} \tag{2.21}$$

Esta cantidad es igual a la tasa máxima de crecimiento de *f*(x, y) por unidad de distancia en la dirección de $\vec{\nabla} f$. En la práctica aproximaremos el gradiente por la suma de los valores absolutos de sus componentes [22]:

$$\left|\vec{\nabla} f(x,y)\right| \approx \left|\frac{\partial f}{\partial x}\right| + \left|\frac{\partial f}{\partial y}\right| \ . \tag{2.22}$$

Esta aproximación, aunque no es demasiado fina, mantiene una relación de semejanza entre la magnitud del gradiente en dos puntos diferentes, por lo que cualitativamente sigue resultando útil y es considerablemente más fácil de implementar, especialmente si usamos un hardware dedicado en exclusiva, como va a ser nuestro caso.

La dirección del gradiente es también importante. Supongamos que $\alpha(x,y)$ representa el ángulo direccional de $\vec{\nabla} f$ en el punto (*x*, y). Del análisis vectorial tenemos:

$$\alpha(x,y) = \tan^{-1}\left(\frac{\left(\frac{\partial f}{\partial y}\right)}{\left(\frac{\partial f}{\partial x}\right)}\right), \tag{2.23}$$

donde el ángulo es medido con respecto al eje *x*. Es una herramienta útil para unir puntos frontera que hayan sido detectados usando el gradiente, en aquellos casos en los que la existencia de ruido en la imagen no permita definir completamente la frontera de un objeto, apareciendo discontinuidades en los bordes.

Ahora vamos a aproximar las derivadas parciales de manera que puedan sernos útiles al tratar con imágenes resultantes de un proceso de digitalización, sustituyendo las variables continuas, *x* e *y*, por los índices discretos *i* y *j*. Si llamamos *G* [*f* (*i*, *j*)] a la aproximación de la magnitud del vector gradiente, tendríamos la expresión:

$$G[f(i,j)] = \left[\left(\Delta f / \Delta i\right)^2 + \left(\Delta f / \Delta j\right)^2\right]^{1/2}, \tag{2.24}$$

En la ecuación anterior se observa cómo el cálculo de la magnitud del gradiente se basa en la obtención de las aproximaciones a las derivadas parciales continuas en el dominio discreto, $\Delta f / \Delta i$ y $\Delta f / \Delta j$, para cada píxel de la imagen. Vamos a calcular estas aproximaciones.

$$I = \begin{pmatrix} I_1 & I_2 & I_3 \\ I_4 & I_5 & I_6 \\ I_7 & I_8 & I_9 \end{pmatrix} \tag{2.25}$$

Observamos la región 3x3 de la ecuación (2.25). En ella, I_5 representa el nivel de gris en el punto (*x*, y) y el resto representa el vecindario de ese punto.

Si usamos una aproximación por diferencias[7], obtendríamos, para el píxel I_5:

$$G[f(i,j)] \approx \left[(I_5 - I_8)^2 + (I_5 - I_6)^2\right]^{1/2}, \tag{2.26}$$

que sigue siendo una expresión difícil de implementar. Sin embargo, podemos conseguir un resultado muy parecido usando valores absolutos [22]:

$$G[f(i,j)] \approx |I_5 - I_8| + |I_5 - I_6| \tag{2.27}$$

Otro método posible sería usar la aproximación por diferencias cruzadas:

$$G[f(i,j)] \approx \left[(I_5 - I_9)^2 + (I_6 - I_8)^2\right]^{1/2} \tag{2.28}$$

[7] Se ha tomado $\Delta i = \Delta j = 1$ en esta aproximación. En caso de considerar otros valores para estos incrementos tendría lugar un efecto de escalado.

o su equivalente en valores absolutos:

$$G[f(i,j)] \approx |I_5 - I_9| + |I_6 - I_8| \tag{2.29}$$

Las expresiones dadas por (2.27) y (2.29) pueden ser implementadas usando máscaras 2x2, como muestran las siguientes ecuaciones:

$$D_{Dv} = \begin{pmatrix} 1 & 0 \\ -1 & 0 \end{pmatrix} \tag{2.30}$$

$$D_{Dh} = \begin{pmatrix} 1 & -1 \\ 0 & 0 \end{pmatrix} \tag{2.31}$$

$$D_{R-45} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{2.32}$$

$$D_{R45} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \tag{2.33}$$

En las ecuaciones anteriores vemos la implementación en forma de máscara 2x2 de los métodos de aproximación anteriormente descritos. Las máscaras de las ecuaciones (2.30) y (2.31) corresponden a la aproximación por diferencias. Las ecuaciones (2.32) y (2.33) representan a los llamados operadores de gradiente cruzado de Roberts, y usan el método de aproximación por diferencias cruzadas.

El uso de regiones 3x3 para computar el gradiente tiene la ventaja de aumentar el suavizado con respecto a los operadores 2x2, haciendo las operaciones derivativas menos sensibles al ruido, permitiendo que los elementos diagonales que rodean al píxel a procesar sean tenidos en consideración para el cálculo del gradiente. Con esa finalidad surgen los operadores de Prewitt. Los operadores Sobel aparecen para añadir un suavizado adicional, que consiguen ponderando los píxeles centrales de las máscaras con un valor de 2. Las siguientes ecuaciones muestran los operadores de Prewitt:

$$P_H = \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} \tag{2.34}$$

$$P_V = \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix} \tag{2.35}$$

Los operadores Sobel se detallan a continuación:

$$S_H = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \tag{2.36}$$

$$S_V = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \tag{2.37}$$

Supongamos que usamos los operadores Sobel para computar cada una de las componentes de la aproximación del vector gradiente, es decir, G_i y G_j. Obtendríamos como respuesta:

$$G_i = (I_7 + 2I_8 + I_9) - (I_1 + 2I_2 + I_3) \tag{2.38}$$

y

$$G_j = (I_3 + 2I_6 + I_9) - (I_1 + 2I_4 + I_7)\ . \tag{2.39}$$

Finalmente, la aproximación de la magnitud del gradiente se hallaría:

$$G[f(i, j)] \approx |G_i| + |G_j|\ . \tag{2.40}$$

Otro de los métodos de diferenciación usados en el procesamiento digital de imágenes es el laplaciano. El laplaciano de una función bidimensional, *f*(*x*, *y*), es una derivada de segundo orden definida como sigue:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{2.41}$$

Siguiendo un razonamiento similar al utilizado en el cálculo de la aproximación del gradiente, se obtiene, una vez realizado el proceso de digitalización de la imagen, la aproximación al laplaciano, que denotaremos como $L[f(i, j)]$, cuya implementación en forma digital se muestra a continuación [22]:

$$L[f(i, j)] = 4I_5 - (I_2 + I_4 + I_6 + I_8), \tag{2.42}$$

expresión que da lugar a una máscara de la forma:

$$L = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}, \tag{2.43}$$

El requerimiento básico en la definición digital del laplaciano es que el coeficiente asociado al píxel central (I_5) sea positivo y los asociados al resto sean negativos. Debido a que el laplaciano es derivativo, la suma de coeficientes ha de ser cero.

Posee la característica de producir un efecto difusivo al hacer convolucionar la máscara mostrada en la ecuación (2.43) (también conocida como máscara de

difusión) sobre una imagen dada. La respuesta de la máscara del laplaciano será nula donde el píxel evaluado y los de su entorno tengan el mismo valor.

Aunque el laplaciano responde a cambios en la intensidad, apenas se usa para detección de bordes por varias razones:

- Es inaceptablemente sensible al ruido, ya que es una derivada de 2º orden.
- Produce bordes dobles.
- Es incapaz de detectar la dirección del borde.

Es por ello por lo que el laplaciano juega un papel secundario en la detección, usándose para establecer si un píxel se encuentra en la cara clara u oscura de un borde. Un uso más general del laplaciano es encontrar la localización de los bordes usando su propiedad de su cruce por cero en la posición de la discontinuidad, que sí es una alternativa real al gradiente, sobre todo en casos en que el borde se encuentre borroso o cuando hay un alto contenido de ruido presente. El problema es que este método conlleva un mayor coste computacional, que puede resultar crítico en aplicaciones en tiempo real.

2.2.2 Simulación de redes dinámicas complejas

A veces resulta interesante definir algún tipo de procesamiento de la imagen como el resultado de la evolución de la dinámica de una red cuyos nudos representan cada uno de los píxeles de la imagen. Un modelo adecuado para describir este tipo de comportamientos es el de las redes neuronales celulares (CNN's) [23]. Se trata de una red de procesadores dinámicos, no lineales, cuyas interacciones, de carácter local, se realizan mediante señales continuas en amplitud y en el tiempo. La conectividad restringida facilita su implementación electrónica. Considerando una CNN desde la perspectiva del procesamiento de señal, la operación de cada celda de la red puede describirse a partir de tres variables:

- Entrada de la celda: $u_{ij}(t)$, que representa la excitación externa.
- El estado de la celda: $x_{ij}(t)$, que da una idea de la energía de la celda en función del tiempo[8].
- Salida de la celda: $y_{ij}(t)$, obtenida a partir del estado mediante una transformación no-lineal:

$$y_{ij} = f\left(x_{ij}\right), \tag{2.44}$$

Para describir el comportamiento de una CNN partimos del modelo de Chua-Yang [23], en el que la evolución de cada celda de la red viene dada por:

$$\tau \frac{\partial x_{ij}(t)}{\partial t} = -x_{ij}(t) + z_{ij} + \sum_{k=-r}^{r}\sum_{l=-r}^{r}[a_{kl} \cdot y_{(i+k)(j+l)} + b_{kl} \cdot u_{(i+k)(j+l)}], \tag{2.45}$$

[8] La variable temporal puede ser continua, representada por *t*, o discreta, representada por *n*. En este último caso, las señales sólo son válidas en determinados instantes dentro de un conjunto discreto de valores, $t=nT$, donde $n= 0, 1, 2, 3 \ldots$

donde:

$$y_{ij} = f\left(x_{ij}\right) = \frac{1}{2}\left(\left|x_{ij}+1\right| - \left|x_{ij}-1\right|\right), \tag{2.46}$$

En la ecuación (2.45) aparece τ, conocida como constante de tiempo de la red, los a_{ij} y b_{ij} son coeficientes de **A** y **B,** que son operadores de interacción entre celdas de la red, y z_{ij}, que es un término de *offset*.

El diagrama esquemático del modelo se muestra a continuación:

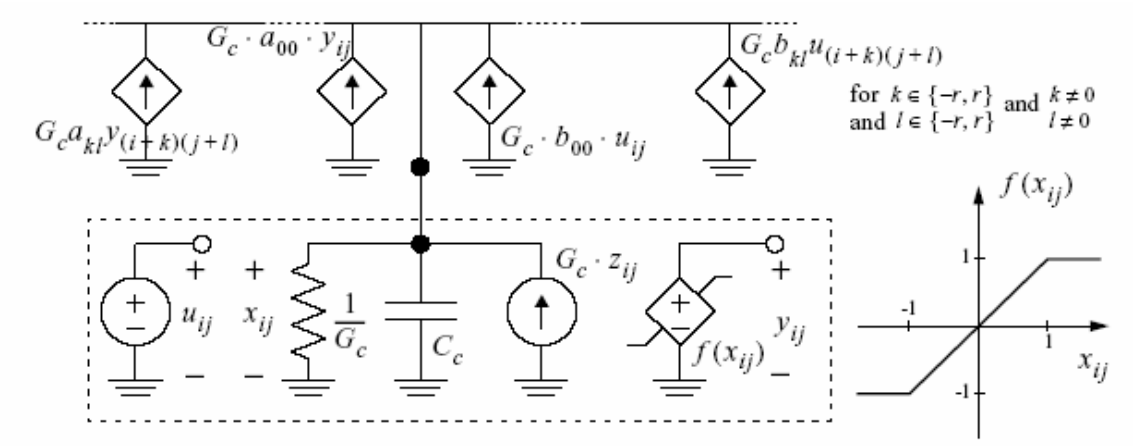


Figura 2.5. Modelo de CNN propuesto por Chua y Yang [23]

En la figura anterior las variables de entrada, estado y salida de la celda están representadas por las tensiones u_{ij}, x_{ij} y y_{ij}, respectivamente. Las contribuciones de los vecinos y la propia realimentación de la celda se realiza mediante corrientes inyectadas por fuentes controladas por tensión (*voltage controlled current sources*, VCCS), cuyas transconductancias son proporcionales a los valores nominales de los pesos de conexión. Estas corrientes se integran en el condensador de estado C_c. El término de pérdidas que encontramos en la ecuación (2.45) aparece aquí como la resistencia lineal $1/G_c$, que hace las funciones de transconductancia de normalización para las VCCS. La constante de tiempo de la red se define mediante $\tau = C_c/G_c$. Una fuente de intensidad en DC implementa el término de *offset* o de *bias,* o *umbral de corriente* (z_{ij}). Los operadores de interconexión son lineales, y, puesto que están restringidos a un radio finito, los operadores **A** y **B**, tienen forma matricial. Comúnmente se les denomina máscaras de conexión o *templates*. De modo que tenemos un template de realimentación, **A**, y otro de control, **B**. Para un radio de vecindad unitario:

$$A = \begin{pmatrix} a_{-1,-1} & a_{-1,0} & a_{-1,1} \\ a_{0,-1} & a_{0,0} & a_{0,1} \\ a_{1,-1} & a_{1,0} & a_{1,1} \end{pmatrix} \quad \text{y} \quad B = \begin{pmatrix} b_{-1,-1} & b_{-1,0} & b_{-1,1} \\ b_{0,-1} & b_{0,0} & b_{0,1} \\ b_{1,-1} & b_{1,0} & b_{1,1} \end{pmatrix}, \tag{2.47}$$

Un caso concreto de comportamiento dinámico que puede describirse mediante las ecuaciones de una red neuronal celular es la difusión de las tensiones en los nudos de una red resistiva. Un esquema simplificado de un nudo de la red sería de la forma:

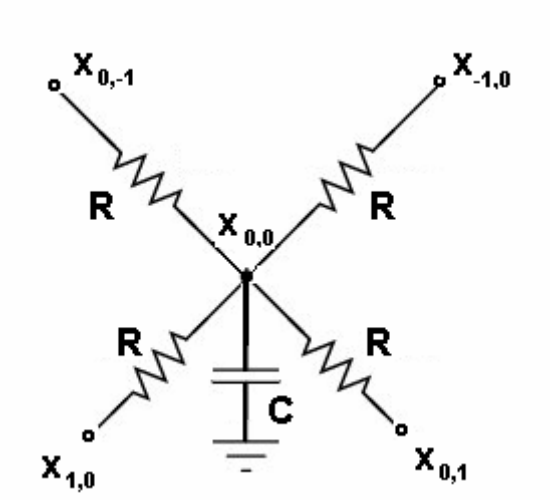


Figura 2.6. Nudo de red resistiva

Si calculamos la intensidad eléctrica entrante en el condensador de la figura anterior, según las leyes de Kirchoff, obtendríamos la siguiente ecuación, que es una versión simplificada de (2.45):

$$\frac{Vx_{1,0}-Vx_{0,0}}{R}+\frac{Vx_{-1,0}-Vx_{0,0}}{R}+\frac{Vx_{0,1}-Vx_{0,0}}{R}+\frac{Vx_{0,-1}-Vx_{0,0}}{R}=C\frac{dVx_{0,0}}{dt} \qquad (2.48)$$

Hacemos $\tau = RC$ y $P = Vx_{1,0}+Vx_{-1,0}+Vx_{0,-1}+Vx_{0,1}-4Vx_{0,0}$. Se observa, desde el punto de vista del procesamiento de imágenes, que P se corresponde con la respuesta de la máscara usada para computar el laplaciano, mostrada en la ecuación (2.43), por lo que estaremos llevando a cabo una difusión de la imagen original. Para aproximar la derivada realizamos una simulación Forward-Euler, así tendremos:

$$\tau\frac{Vx_{0,0}(n+1)-Vx_{0,0}(n)}{\Delta t}=P; \qquad (2.49)$$

Despejando el nuevo valor tomado en el nodo que estamos considerando:

$$Vx_{0,0}(n+1)=Vx_{0,0}(n)+\frac{\Delta t}{\tau}P\ ; \qquad (2.50)$$

De la expresión anterior hemos de extraer, traduciendo al ámbito del procesamiento digital de imágenes, que el valor del píxel procesado vendrá dado por el que tenía antes de ser tratado, añadiéndole la aportación ponderada de la diferencia entre el propio píxel y sus vecinos, dada por P. El valor de la aportación dependerá bastante de la constante τ y del tiempo Δt, que será el tiempo que dejamos correr la difusión en cada iteración, o sea, el paso de integración.

3. ARQUITECTURA DEL SISTEMA DE VISIÓN ON-CHIP

3.1 Descripción del sistema

El sistema de visión tiene como misión realizar un procesamiento digital de imágenes, llevando a la práctica los fundamentos teóricos expuestos en el apartado anterior. Este sistema consta de una arquitectura convencional con un microprocesador y con los elementos periféricos necesarios para la adquisición y almacenamiento de datos y las comunicaciones con el entorno de programación y con la fuente de imágenes. Además, y con el fin de liberar al microprocesador de las tareas de procesamiento de imágenes en bajo nivel, hemos diseñado un coprocesador visual especializado que funciona de manera autónoma, a demanda del microprocesador, como un periférico más.

El sistema de visión incorpora en un único chip todos los componentes necesarios para el procesamiento y transmisión de imágenes, salvo el sensor o fuente de imágenes, los cuales describimos a continuación:

- *Procesador de propósito general*, funcionando como unidad microcontroladora (MCU), y por tanto encargada de gestionar los buses del sistema y el intercambio de datos con el exterior.
- *Coprocesador visual*, encargado de procesar las imágenes mediante el uso de máscaras espaciales y otros operadores especialmente diseñados para ese propósito.
- *Memoria RAM de doble puerto (DPRAM)*, que será la memoria de datos, es el lugar donde la MCU almacena imágenes recibidas y con la que el coprocesador intercambia datos de entrada y/o salida.

La siguiente figura muestra un posible esquema del sistema de visión:

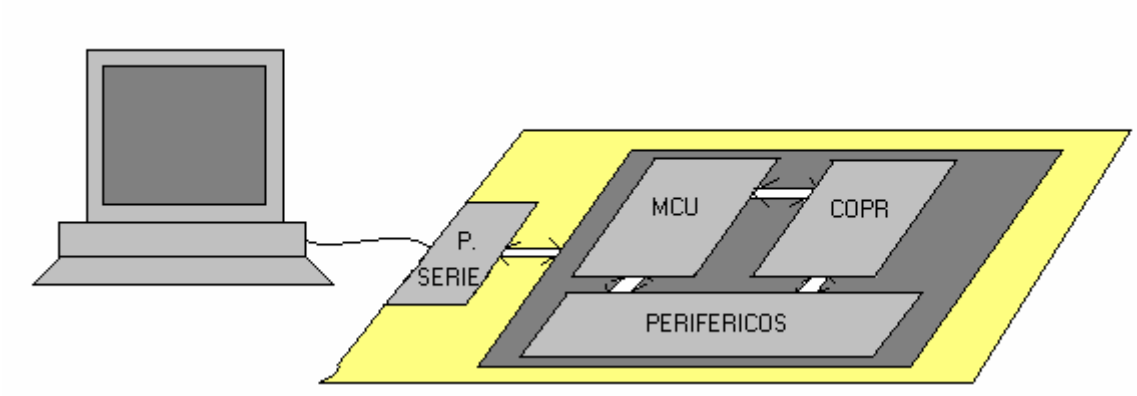


Figura 3.1 Sistema de visión controlado por un PC.

El esquema aquí propuesto se encuentra controlado por un PC, que actuaría como fuente de imágenes, recibidas por la MCU a través del puerto serie, almacenadas en la DPRAM, y procesadas por el coprocesador visual. La tarea

concreta de procesamiento que realizaría el coprocesador le vendría encomendada por la MCU, que es el dispositivo maestro dentro del sistema de visión.

3.2 Unidad microcontroladora (MCU)

El sistema de visión que vamos a diseñar está basado en el microprocesador de código abierto Aquarius, obtenido de OpenCores.org, página web dedicada a proporcionar módulos digitales (Cores) de manera libre y sin ánimo de lucro. Expondremos en este apartado las características del microprocesador.

La elección de este sistema frente a otros que hemos tanteado se ha debido principalmente a la completa accesibilidad que hemos tenido a los recursos del mismo, es decir:

- Su código en lenguaje HDL, perfectamente comentado,
- Amplia documentación adjuntada,
- Diferentes ejemplos,
- Herramientas necesarias para la programación y desarrollo de aplicaciones, así como para la simulación de las mismas,
- Cumple con una interfaz muy simple y novedosa dentro de los SoCs (System on Chip), llamada interfaz WISHBONE, que detallaremos cuando hablemos de los buses del sistema.

Un ejemplo de plataformas para el desarrollo de sistemas empotrados (embedded systems) que no han sido elegidos puede ser el microprocesador Microblaze, de Xilinx. Un sistema muy potente y fácilmente programable, pero con el importante inconveniente de la privacidad del código, que hacía imposible una adecuada aproximación al estudio a fondo del mismo, encontrando trabas a la hora de simular el microprocesador. El inconveniente de la privacidad del código es fundamental, ya que estamos implementando un coprocesador, para lo que resulta indispensable saber cómo es el sistema maestro internamente, para saber cómo hay que intercambiar información con él de manera óptima. Es por ello Aquarius una elección, al menos, interesante.

3.2.1 Descripción de Aquarius

Aquarius es un procesador de código abierto [24] con juego de instrucciones reducido (RISC) y que puede ejecutar instrucciones SuperH-2 ISA (Instruction Set Architecture) [25]. Este juego de instrucciones ha sido elegido por su gran popularidad, ya que ha sido usado en una gran variedad de dispositivos, tales como grabadores de DVD, robots, PDA's, FAX, etc. [25] lo que ha permitido que tenga entornos de desarrollo de software, como un compilador cruzado de C. Además, el compilador GNU es muy fácil de obtener, como explicaremos más adelante.

Otra razón de elegir SuperH-2 ISA es que evita excepciones demasiado complejas ya el tamaño de sus instrucciones es de tan sólo 16 bits, lo que nos da ventajas como la facilidad de diseño. La simpleza estructural de SuperH-2, el escaso consumo de puertas lógicas y el reducido tamaño del código objeto compilado de programas fuente en C, consecuencia de tener instrucciones de

tan solo 16bit, son otras de las razones que invitan a escoger este juego de instrucciones.

Aquarius sigue las especificaciones de una interfaz WISHBONE [26], como detallaremos en el apartado relativo a los buses del sistema, lo que le permite conectarse directamente a otros bloques de IP (propiedades intelectuales) que cumplan dichas especificaciones, facilitando la integración de SoCs. (Systems On Chip)

3.2.1.a Estructura de la CPU de Aquarius

El diagrama de bloques del sistema se muestra a continuación:

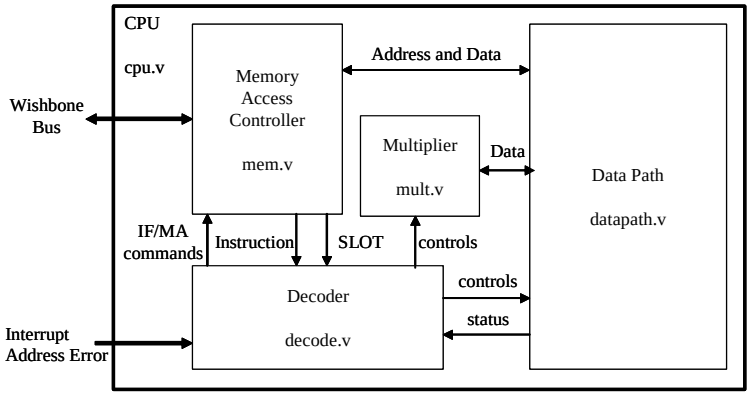


Figura 3.2 Diagrama de bloques de Aquarius [24]

En la figura aparecen las diferentes unidades que integran la unidad central de proceso (CPU) del sistema y que describimos a continuación:

Unidad de decodificación

La unidad de decodificación es la controladora fundamental de la CPU. Recibe la instrucción recogida por el controlador de acceso a memoria, decodifica campos de bits y decide qué operaciones van a ser llevadas a cabo a continuación. Esta unidad detecta atascos en la estructura pipeline (explicado con más detalle más adelante dentro de este apartado) y hace que cada unidad de la CPU sea controlada adecuadamente. Además, controla instrucciones de más de un ciclo de reloj, secuencias de excepción, interrupciones y errores de direccionamiento. Se corresponde con el fichero Verilog *decode.v*.

Controlador de acceso a memoria

El controlador de acceso a memoria (Memory Access Controller) maneja la recogida de instrucciones y los accesos de lectura / escritura. Sus operaciones están controladas completamente por la unidad de decodificación (Decoder Unit). Intercambia datos y direcciones con la unidad de encaminamiento de datos (Data Path Unit) y envía campos de bits de las instrucciones recogidas a la unidad de decodificación.

Aquarius asume que el bus WISHBONE no posee buses separados de datos e instrucciones, es decir, no es un microprocesador con arquitectura Harvard, por lo que es posible que haya contención del bus. El controlador de acceso a memoria maneja cada contienda con cuidado e informa del atasco del pipeline a la unidad de decodificación. También se encarga de chequear el correcto fin de cada ciclo de bus a través de la señal WISHBONE ACK. Se corresponde con el fichero *mem.v*.

Data Path

La unidad llamada Data Path (*datapath*.v), que contiene la gran mayoría de registros del sistema, con pocas excepciones, tiene la función de, a partir de las señales de control procedentes de la unidad de decodificación, realizar el direccionamiento apropiado de los datos dentro de la CPU. Como hemos dicho, en ella residen los registros generales (R0-R15), el registro base global (GBR), el registro de estado (SR), el registro vector base (VBR), el registro de proceso (PR) y el contador de programa (PC). Esta unidad incorpora recursos de operación como la ALU, unidad de desplazamiento, divisor, comparador, registros temporales, etc. La unidad aritmético-lógica (ALU) puede ser usada en el cálculo de direcciones destino y otras operaciones básicas para el funcionamiento de la CPU. El divisor, el comparador, los registros temporales y la unidad de desplazamiento realizan operaciones en las que los operandos son modificados.

Unidad de multiplicación

La unidad de multiplicación (*mult.v*) posee un multiplicador de 32bit x 16bit y sus circuitos de control. Es capaz de ejecutar una multiplicación 16bit x 16bit en un ciclo de reloj, y una 32bit x 32bit en dos ciclos de reloj. Esta unidad contiene los registros de multiplicación y acumulación (MACH/MACL). Estos registros no se limitan a guardar el resultado final de una operación aritmética, sino que sirven de registros temporales para mantener el resultado parcial de 48bit resultante de una multiplicación 32bit x 32bit. Además, al ejecutar la instrucción MAC.L estos registros no son borrados previamente a una nueva operación, como ocurre con una operación de multiplicado normal (MULS.L), permitiendo almacenar el resultado parcial en MACH/MACL.

Como referimos anteriormente, los nombres de los archivos que aparecen en cada uno de los bloques de la figura 3.2, que se encuentran descritos en formato Verilog, pueden ser descargados libremente de la página *http://www.opencores.org/cvsweb.shtml/Aquarius*, concretamente en el directorio llamado *verilog*. Los ficheros contenidos en dicho directorio conforman el sistema microprocesador completo, y son los siguientes:

Archivos para el banco de pruebas

timescale.v	Escala temporal. Todos los ficheros la incluyen.
test.v	El banco de pruebas.
top.v	La capa más alta del sistema.

Archivos que conforman la MCU (Unidad de Micro Controlador)

top.v	Capa más alta de la MCU.
memory.v	ROM (8KB) y RAM (8KB) para simulación verilog.
rom.v	Descripción de la ROM (8KB) a partir de formato S[9].
pio.v	Interfaz de E/S paralela.
memory_fpga.v	ROM (8KB) y RAM (8KB) para configuración FPGA. Toda su área puede ser inicializada por especificaciones INIT.
uart.v	UART (Transmisor / Receptor Asíncrono Universal).
sasc_brg.v	Generador de la tasa de baudios.
sasc_top.v	Cuerpo de UART.
sasc_fifo4.v	FIFO de 4 elementos del buffer de la UART.
sys.v	Controlador del sistema. Manejo de interrupciones.
lib.v	Puerta para parada del reloj (instrucción SLEEP).
lib_fpga.v	"lib.v" para la configuración de la FPGA.
cpu.v	Capa más alta de la CPU.

Archivos que conforman la CPU

cpu.v	Capa más alta de la CPU.
datapath.v	Data Path
register.v	Registros de propósito general R0-R15.
decode.v	Decodificador de instrucciones.
mem.v	Controlador de acceso a memoria.
mult.v	Multiplicador.
defines.v	Parámetros constantes referenciados desde la CPU.

[9] Formato del código ensamblador de las instrucciones SuperH-2 ISA.

Señales de entrada / salida de la CPU de Aquarius

Las señales de entrada y salida de la CPU de Aquarius se muestran en la siguiente tabla:

Clase	Nombre de señal	Dirección	Significado	Notas
Señales Del Sistema	CLK	Entrada	Reloj sistema	
	RST	Entrada	Reset	
Señales Del Bus WISHBONE	CYC_O	Salida	Salida Ciclo	
	STB_O	Salida	Salida Strobe	
	ACK_I	Entrada	Asentimiento de dispositivo	
	ADR_O[31:0]	Salida	Dirección de salida	
	DAT_I[31:0]	Entrada	Dato leído	
	DAT_O[31:0]	Salida	Dato escrito	
	WE_O	Salida	Habilitación de escritura	
	SEL_O[3:0]	Salida	Selección de byte	
	TAG0_I (IF_WIDTH)	Entrada	Ancho de Recogida	
Evento Hardware (interrupción)	EVENT_REQ_I[2:0]	Entrada	Petición de Evento	
	EVENT_INFO_I[11:0]	Entrada	Información de Evento	
	EVENT_ACK_O	Salida	Asentimiento de Evento	
SLEEP	SLP	Salida	Pulso de Sleep	

Tabla 3.1 Señales de entrada / salida de la CPU de Aquarius [24]

La arquitectura pipeline de Aquarius

La siguiente figura detalla la arquitectura pipeline de Aquarius, mostrando sus diferentes fases:

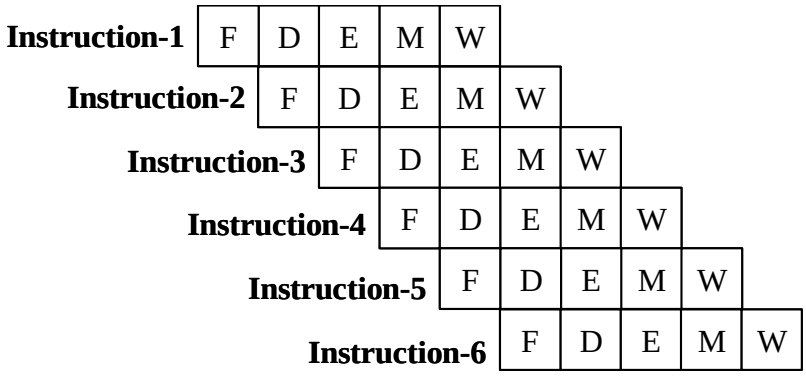


Figura 3.3 Estructura del pipeline de Aquarius [24]

La estructura consta de 5 etapas, no teniendo que aparecer necesariamente todas ellas en la ejecución de una instrucción en concreto. Detallamos las diferentes etapas:

IF : Instruction Fetch (“F”)

Es la etapa de recogida de instrucción. En ella se recoge el código de la misma de memoria. El ancho de cada instrucción es de 16bit, así, si el ancho del bus de datos es de 32bit y los dos bits menos significativos de la dirección de memoria a la que accedemos tienen el valor 2’b00 (ambos a cero, en sintaxis Verilog), podremos recoger dos instrucciones a la vez. Si el ancho del bus de datos es menor o si los dos bits menos significativos de la dirección accedida no valen 2’b00 sólo recogeremos una instrucción, de acuerdo con lo estipulado por las especificaciones WISHBONE [26].

ID : Decode (“D”)

El código de la instrucción recogida es decodificado y se controla el funcionamiento de toda la CPU. La etapa ID es la más importante debido a que todas las operaciones a realizar en cada bloque de la CPU se definen en esta etapa. Activa diferentes señales de control que sirven al resto de etapas del pipeline, EX, MA y WB.

EX : Execute (“E”)

Fase de ejecución. De acuerdo con las señales de control procedentes de la etapa ID, EX ejecuta la operación registro-registro (transferencia de datos entre los mismos) o calcula la dirección para la siguiente fase MA.

MA : Memory Access (“M”)

De acuerdo con las señales de control procedentes de la etapa ID, MA lee/escribe datos de/en memoria. La CPU de Aquarius no posee una arquitectura Harvard, por tanto, unas IF y MA simultáneas pueden provocar contención del bus. En este caso, MA tiene mayor prioridad, por tanto IF sería detenida por MA.

WB : Write Back (“W”)

De acuerdo con las señales de control procedentes de ID, la etapa WB rescribe en el registro Rn los datos leídos de memoria. Aparece al final del pipeline en instrucciones de carga de datos de memoria.

Veamos unos ejemplos, extraídos de la guía de usuario de Aquarius [24], de ejecución de algunas instrucciones para observar el funcionamiento de la estructura pipeline de Aquarius:

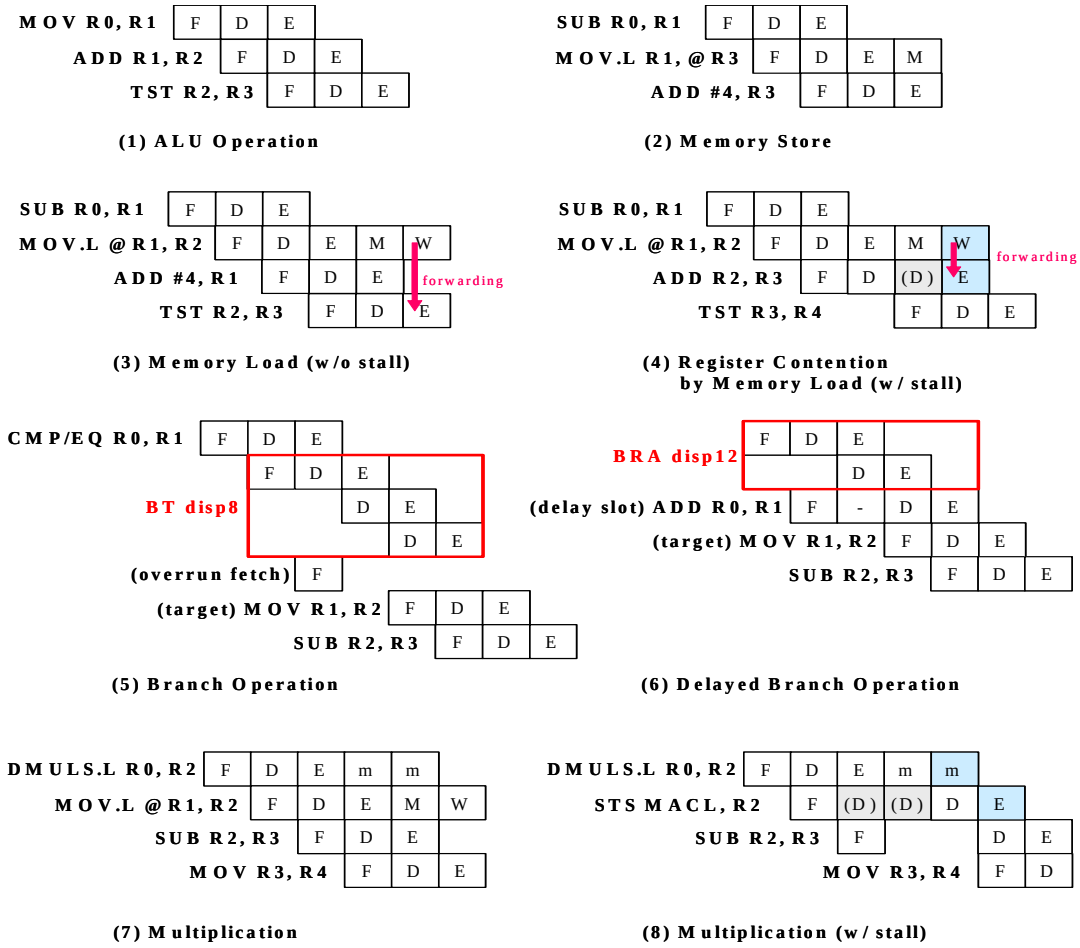


Figura 3.4 Ejemplos de ejecución de instrucciones en Aquarius [24]

En la figura anterior se pone de manifiesto que no todas las instrucciones necesitan 5 fases para su ejecución. Comentamos las operaciones mostradas con anterioridad:

- **(1) Operación ALU:** La instrucción de la operación de registro a registro tiene sólo 3 etapas; IF, ID y EX. La operación de registro a registro se ejecuta en la etapa EX, incluyendo la lectura del registro, la operación ALU y la escritura en registro.
- **(2) Almacenado en memoria:** Esta instrucción posee 4 fases; IF, ID, EX y MA. La dirección de acceso a memoria se calcula en la etapa EX, en la que también se prepara la escritura del dato.
- **(3) Carga desde memoria:** Esta instrucción consta de 5 etapas; IF, ID, EX, MA y WB. La dirección de acceso a memoria se calcula en la etapa EX. El dato cargado es almacenado en un registro en la etapa WB. Si el registro a ser reescrito no es el mismo que el registro que va a ser usado en la siguiente instrucción, no habrá contención del registro, por lo que el pipeline podrá fluir sin atascarse. La etapa EX en la última instrucción, que usa el dato reescrito en la WB, puede ser ejecutada al mismo tiempo que la WB gracias al control ejercido por parte de la etapa ID (lo señalado como *forwarding* en la figura).

- **(4) Carga desde memoria con contención de registro:** Si el registro a rescribir es el mismo que el usado en la instrucción siguiente, tiene lugar la contención del registro. La etapa ID de la instrucción siguiente es congelada.
- **(5) Operación de salto:** La operación de salto tiene ciclos múltiples. En el recuadro señalado en la figura 3.4 (5), pueden verse 3 pipelines. Esto quiere decir que la instrucción BT se ejecuta en 3 ciclos. Generalmente, las instrucciones de ciclos múltiples consisten en múltiples pipelines. En el caso de la instrucción BT, el 1º pipeline, calcula la dirección del objetivo del salto (en el dibujo, "target"), el 2º pipeline resuelve la recogida de la instrucción objetivo del salto ("target") e incrementa PC (contador de programa), y el 3º pipeline lleva a cabo la recogida de la instrucción posterior al objetivo del salto e incrementa PC. La instrucción previa a BT ha llevado a cabo una recogida de instrucción, pero el código recogido será sobrescrito por la etapa IF (del objetivo del salto) realizada por el 2º pipeline de BT antes de ser enviado a la etapa ID de la instrucción objetivo. Esta instrucción extra es conocida como "overrun fetch". El código recogido por esta instrucción extra es ignorado.
- **(6) Salto con retraso:** El salto con retraso consta de 2 pipelines. El 1º calcula la dirección del objetivo del salto ("target"), el 2º lleva a cabo la recogida de la instrucción objetivo del salto e incrementa PC. La etapa IF de la instrucción colocada en la posición de retraso ("delay slot", en la figura), la cual ha sido realizada por la instrucción previa al salto con retraso, no desaparece (no es sobrescrita), por lo que la instrucción colocada en la posición de retraso ("delay slot") es ejecutada correctamente antes de la instrucción objetivo del salto.
- **(7) Multiplicación:** Las instrucciones relacionadas con la multiplicación poseen una etapa llamada precisamente de multiplicación, ("m") en la cola del pipeline. Si el registro de resultados, MACH/MACL, no entra en conflicto con instrucciones posteriores, no habrá atasco en el pipeline.
- **(8) Multiplicación con contención de registro:** Si los registros de resultados, MACH/MACL, entran en conflicto con las instrucciones posteriores, tiene lugar el atasco en el pipeline.

3.2.1.b Arquitectura del bus WISHBONE

La arquitectura para la interconexión de sistemas empotrados, WISHBONE [26], permite una fácil integración e dispositivos en un SoC. Define un esquema estándar de interconexión con un bajo consumo de puertas lógicas, lo que simplifica la conexión entre diferentes dispositivos. Algunas de sus características principales son:

- Interfaces hardware sencillas y compactas entre IP Cores que requieren muy pocas puertas lógicas.

- Soporta diversos métodos de interconexión: punto a punto, bus compartido, etc.
- Incluye protocolos clásicos de transferencia de datos a través de un bus: ciclos de lectura y escritura, ciclos de transferencia de bloques, ciclos RMW (lectura de un dato de memoria, procesado del mismo y posterior escritura del mismo en memoria), etc.
- Los tamaños del bus de datos, bus de direcciones y de los operandos son modulares, lo que permite diseñar estructuras escalables.
- Soporta transferencias que duran un sólo ciclo de reloj.
- El protocolo del saludo (*handshaking protocol*) permite a cada IP core regular su velocidad de transferencia de datos.
- Permite el uso de etiquetas definidas por el usuario (tags), que permiten añadir información adicional al bus de direcciones, datos o al ciclo del bus, como por ejemplo: paridad, vectores de interrupción, operaciones de control sobre la caché, etc.
- Posibilidad de organizar una arquitectura maestro/esclavo (MASTER/SLAVE), para el diseño de sistemas muy flexibles. (Figura 3.5)
- Implementada con éxito en diferentes tecnologías hardware: FPGA, ASIC, etc.
- Independencia de la herramienta de síntesis.

La arquitectura maestro/esclavo se muestra a continuación con sus señales típicas. Vemos que hay un módulo externo llamado *SYSCON*, que se encarga de generar el reloj y el reset que irán al resto de dispositivos del sistema como entrada.

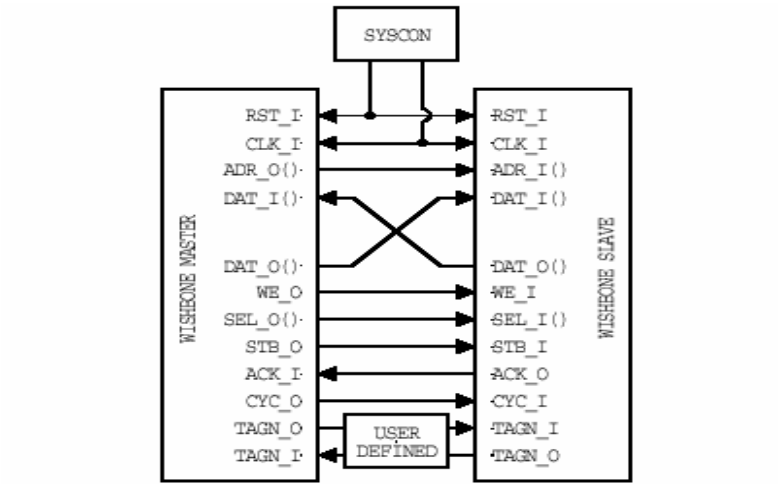


Figura 3.5 Conexión estándar punto a punto [26]

Algunas de las señales importantes son:

- **CYC_X**, que indica que hay un ciclo de bus válido cuando está activa,
- **ACK_X**, que es la terminación normal de un ciclo de bus,
- **STB_X**, que indica una transferencia de datos válida,

- **SEL_X**, que selecciona cuáles de los bytes se van a recoger del bus o van a ser puestos en él,
- **TAGx_X**, etiquetas definidas por el usuario,
- **WE_X**, que muestra si el ciclo actual es de lectura o escritura y las entradas y salidas de datos y direcciones.

NOTA: Todas las señales deben llevar un sufijo que especifique si es de entrada o salida al dispositivo. Es decir, no habrá una señal llamada WE, sino que será WE_O (salida) o WE_I (entrada). Además, todas las señales en una interfaz tipo WISHBONE han de ser activas a nivel alto.

Vemos cómo sería el protocolo del saludo:

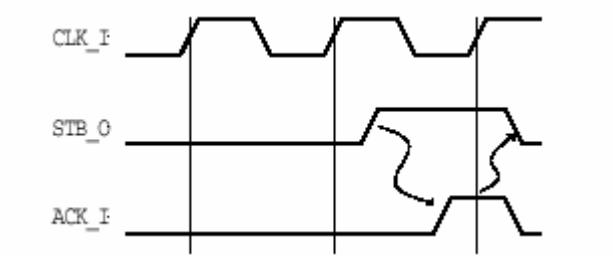


Figura 3.6 Ejemplo de *handhaking protocol* [26]

Todos los ciclos de bus usan este protocolo entre las interfaces del maestro y del esclavo.

Como puede verse en la figura, el maestro activa la señal STB_O cuando está listo para transferir datos. Esta señal permanece activada hasta que el esclavo activa una de las señales de terminación del ciclo de bus, esto es, ACK_I, ERR_I (en caso de error detectado) o RTY_I (en caso de petición de retransmisión). La señal de terminación del ciclo es muestreada en cada flanco de subida del reloj. Si está activada, Entonces se procede a desactivar STB_O. Esto permite tanto a la interfaz del maestro como a la del esclavo controlar la tasa a la que son transferidos los datos.

La siguiente figura muestra un ejemplo de cómo sería un acceso clásico en lectura en un bus de este tipo, desde el punto de vista del maestro:

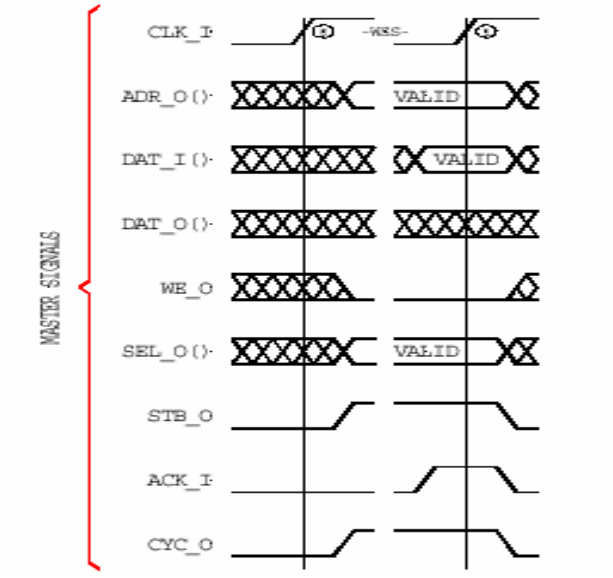


Figura 3.7 Acceso simple en lectura bus WISHBONE [26]

NOTA: En la figura hay detalles que posiblemente no se aprecien con una buena resolución, como unos números rodeados por un círculo al lado de los flancos de subida del reloj. Éstos corresponden al número de flanco mostrado, que en este caso, y en el de la figura siguiente, poseen un valor de 0 y 1, es decir, se trata de dos ciclos consecutivos.

Veamos cómo funciona el protocolo del bus:

1. **Flanco de reloj 0**:
    - El MAESTRO presenta una dirección válida en ADR_O.
    - El MAESTRO pone a cero WE_O para indicar que es un ciclo de lectura.
    - El MAESTRO presenta la señal de selección de byte válido, SEL_O, para indicar dónde espera los datos.
    - El MAESTRO activa la señal CYC_O para indicar el comienzo del ciclo.
    - El MAESTRO activa la señal STB_O para indicar el comienzo de la transferencia de datos.

2. **Entre flancos 0 y 1**:
    - Cada ESCLAVO decodifica las entradas, y el correspondiente ESCLAVO activa ACK_I.
    - El ESCLAVO presenta un dato válido en DAT_I.
    - El ESCLAVO[10] activa la señal ACK_I también en respuesta a STB_O para indicar dato válido.
    - El MAESTRO monitoriza ACK_I y se prepara para leer datos de DAT_I.

3. **Flanco 1**:
    - El MAESTRO guarda en un registro el dato de DAT_I.
    - El MAESTRO desactiva STB_O y CYC_O para indicar el final del ciclo.
    - El ESCLAVO desactiva ACK_I en respuesta a la desactivación de STB_O.

En el ciclo de lectura mostrado antes se observa cómo se lleva a cabo el protocolo del saludo visto antes, y cómo hay que tener en cuenta la señal DAT_I, de entrada al maestro, por tratarse de un acceso de lectura. Por ser un acceso simple STB_O y CYC_O coinciden, es decir, no es un acceso de lectura o escritura en bloque (BLOCK READ/WRITE Cycles), también permitido por el bus WISHBONE.

Vemos un ejemplo de ciclo de escritura, también desde el punto de vista del maestro:

[10] El ESCLAVO puede insertar estados de espera (-WSS-) antes de activar ACK_I para adecuar la velocidad del ciclo a sus necesidades o al protocolo establecido con el MAESTRO. Pueden ser añadidos cuantos ciclos sean requeridos.

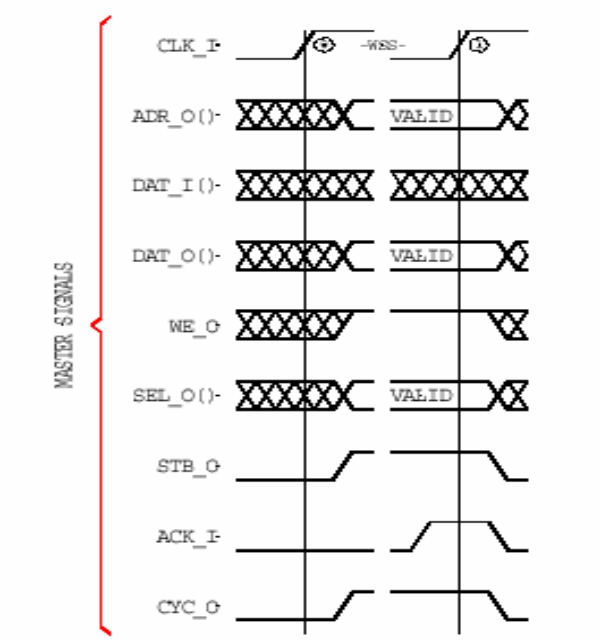


Figura 3.8 Acceso simple en escritura bus WISHBONE [26]

Veamos cómo funciona el protocolo del bus, ahora en escritura:

1. **Flanco de reloj 0**:
   - El MAESTRO presenta una dirección válida en ADR_O.
   - El MAESTRO presenta un dato válido en DAT_O.
   - El MAESTRO activa WE_O para indicar que es un ciclo de escritura.
   - El MAESTRO presenta la señal de selección de byte válido, SEL_O, para indicar dónde envía los datos.
   - El MAESTRO activa la señal CYC_O para indicar el comienzo del ciclo.
   - El MAESTRO activa la señal STB_O para indicar el comienzo de la transferencia de datos.

2. **Entre flancos 0 y 1**:
   - Cada ESCLAVO decodifica las entradas, y el correspondiente ESCLAVO activa ACK_I.
   - El ESCLAVO se prepara para leer el dato de DAT_O.
   - El ESCLAVO[11] activa la señal ACK_I también en respuesta a STB_O para indicar dato válido.
   - El MAESTRO monitoriza ACK_I y se prepara para terminar el ciclo.

3. **Flanco 1**:
   - El ESCLAVO guarda en un registro el dato de DAT_O.
   - El MAESTRO desactiva STB_O y CYC_O para indicar el final del ciclo.

[11] El ESCLAVO puede insertar estados de espera (-WSS-) antes de activar ACK_I para adecuar la velocidad del ciclo a sus necesidades o al protocolo establecido con el MAESTRO. Pueden ser añadidos cuantos ciclos sean requeridos.

- El ESCLAVO desactiva ACK_I en respuesta a la desactivación de STB_O.

3.2.1.c Periféricos de Aquarius

Hablamos ahora de los periféricos internos de Aquarius, además de mostrar ejemplos de ciertas configuraciones posibles con este procesador. Los periféricos que comparten chip con la CPU son fundamentalmente las memorias RAM y ROM, los puertos serie y paralelo, UART y PIO, respectivamente, y el controlador del sistema (System Controller). Vemos a continuación el mapa de memoria original de Aquarius, en el que aparecen referenciados los periféricos que tratamos en este apartado.

Address	Device	Size	Access Cycle	IF Width	Notes
0x00000000-0x00001FFF	ROM	8KB	1cyc	32bit	A
0x00002000-0x00003FFF	RAM	8KB	1cyc	32bit	B
0x00004000-0x0000FFFF	Shadow of 0x00000000-0x00003FFF				
0x00010000-0x00011FFF	ROM	8KB	4cyc	32bit	Shadow of A
0x00012000-0x00013FFF	RAM	8KB	4cyc	32bit	Shadow of B
0x00014000-0x0001FFFF	Shadow of 0x00010000-0x00013FFF				
0x00020000-0x00021FFF	ROM	8KB	1cyc	16bit	Shadow of A
0x00022000-0x00023FFF	RAM	8KB	1cyc	16bit	Shadow of B
0x00024000-0x0002FFFF	Shadow of 0x00020000-0x00023FFF				
0x00030000-0x00031FFF	ROM	8KB	4cyc	16bit	Shadow of A
0x00032000-0x00033FFF	RAM	8KB	4cyc	16bit	Shadow of B
0x00034000-0x0003FFFF	Shadow of 0x00030000-0x00033FFF				
0x00040000-0xABCCFFFF	Shadow of 0x00000000-0x0003FFFF				
0xABCD0000-0xABCD00FF	PIO	256B	4cyc	32bit	
0xABCD0100-0xABCD01FF	UART	256B	4cyc	32bit	
0xABCD0200-0xABCD02FF	SYS	256B	4cyc	32bit	
0xABCD0300-0xFFFFFFFF	Shadow of 0x00000000-0x0003FFFF				

Tabla 3.2 Mapa de memoria de Aquarius [24]

Memorias “en-chip”

El módulo de memoria, *memory.v*, tiene 8Kbytes de ROM y 8Kbytes de RAM. El mapa de memoria mostrado antes indica la distribución de las mismas. Las señales de entrada / salida del módulo de memoria se muestra a continuación.

Clase	Nombre de señal	Dirección	Significado	Notas
Señales del Sistema	CLK	Entrada	Reloj del sistema	
	RST	Entrada	Reset	
Señales del Bus WISHBONE	CE	Entrada	Chip Select (Selección de módulo)	STB
	WE	Entrada	Write Enable	
	SEL[3:0]	Entrada	Selección de byte	
	ADR[13:0]	Entrada	Dirección	
	DATI[31:0]	Entrada	Dato de entrada (Dato escrito por CPU)	
	DATO[31:0]	Salida	Dato de salida (Dato leído)	

Tabla 3.3 Señales de entrada / salida de la memoria interna [24]

La memoria ROM del sistema Aquarius tiene una capacidad de 8Kbytes. En ella, se almacenan programas de aplicaciones que explotarán la funcionalidad del microprocesador. En un próximo apartado detallaremos el proceso de programación de dichas aplicaciones. El fichero Verilog asociado a esta memoria es *rom.v*. Este fichero será útil para simulación.

La memoria RAM de Aquarius también posee una capacidad de 8Kbytes. En ella se escribirán programas de aplicación para su implementación en el dispositivo reconfigurable, una FPGA por ejemplo. El fichero generado no servirá para la simulación, por eso no tiene formato Verilog, es *ram.dat*. Todo esto se explicará más adelante, como dijimos con anterioridad.

Entrada / Salida serie (UART)

La UART (Universal Asynchronous Receiver Transmitter) incluida en Aquarius se encuentra basada en la propiedad intelectual SASC (Simple Asynchronous Serial Communication Device) de OpenCores.org . SASC no cumplía con las especificaciones WISHBONE, por lo que le fueron añadidos una serie de registros para conectarla con el Bus WISHBONE. Todos sus registros se encuentran localizados en direcciones consecutivas de memoria, pudiendo ser accedidos por operandos tamaño byte, word o long, excepto los registros UARTCON y UARTRXD/TXD, que deben ser accedidos por operandos tamaño byte exclusivamente.

Los ficheros Verilog referidos a la UART son los que comienzan con *sasc*, además del llamado *uart.v*.

A continuación se muestran las señales de E/S y los registros de la UART.

Clase	Nombre de señal	Dirección	Significado	Notas
Señales del Sistema	CLK	Entrada	Reloj del sistema	
	RST	Entrada	Reset	
Señales del Bus WISHBONE	CE	Entrada	Chip Select (Selección de módulo)	STB
	WE	Entrada	Write Enable	
	SEL[3:0]	Entrada	Selección de byte	
	DATI[31:0]	Entrada	Dato de entrada (Dato escrito por CPU)	
	DATO[31:0]	Salida	Dato de salida (Dato leído)	
UART	RXD	Entrada	Recepción Dato Serie	
	TXD	Salida	Transmisión Dato Serie	
	CTS	Entrada	Listo para enviar	
	RTS	Salida	Petición de envío	

Tabla 3.4 Señales de entrada / salida de la UART [24]

Los registros:

[UART] Address=0xABCD0100 R/W UARTBG0 (Baud rate Generator Div0)

31(7)	30(6)	29(5)	28(4)	27(3)	26(2)	25(1)	24(0)
B07	B06	B05	B04	B03	B02	B01	B00

[UART] Address=0xABCD0101 R/W UARTBG1 (Baud rate Generator Div1)

23(7)	22(6)	21(5)	20(4)	19(3)	18(2)	17(1)	16(0)
B17	B16	B15	B14	B13	B12	B11	B10

[UART] Address=0xABCD0102 R only UARTCON (TXF=full_o, RXE=empty_o)

15(7)	14(6)	13(5)	12(4)	11(3)	10(2)	9(1)	8(0)
reserved	reserved	Reserved	reserved	reserved	reserved	TXF	RXF

[UART] Address=0xABCD0103 R only / UARTRXD, W only / UARTTXD

7(7)	6(6)	5(5)	4(4)	3(3)	2(2)	1(1)	0(0)
TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0

Tabla 3.5 Registros de la entrada / salida serie (UART) [24]

Describimos los diferentes registros mostrados en la figura precedente:

- **UARTBG0**: Registro usado para la determinación de la tasa de baudios de funcionamiento del dispositivo. Explicado a continuación.
- **UARTBG1**: Registro usado para la determinación de la tasa de baudios de funcionamiento del dispositivo. Explicado a continuación.
- **UARTCON**: Registro de control de la UART. Sólo son de utilidad sus dos bits menos significativos, TxF y RxE, usados para comprobar si la pila de transmisión se encuentra llena y la de recepción vacía. Son de gran utilidad en el funcionamiento normal del dispositivo de comunicación serie.
- **UARTTXD/UARTRXD**: Registro usado para leer el dato recibido, cuando accedemos en lectura, o para escribir el dato a transmitir, cuando accedemos en escritura.

Los registros UARTBG0 y UARTBG1 son los que determinan la tasa de baudios serie. Ambos registros son inicializados al valor 0x00 cuando damos un

reset. La expresión usada para calcular la tasa de baudios se muestra a continuación:

$$BaudRate = \frac{f(CLK)}{4} \times \frac{1}{(BG0+2)\times(BG1)}[bps] \qquad (3.1)$$

Vemos varios ejemplos de configuraciones de tasas de baudios:

Tasa Baudios [bps]	f (CLK) [MHz]	UARTBG0	UARTBG1	Notas
1200	20	0x12 (18)	0xCF (207)	
2400	20	0x12 (18)	0x67 (103)	
4800	20	0x12 (18)	0x33 (51)	
9600	20	0x12 (18)	0x19 (25)	

Tabla 3.6 Ejemplos de configuraciones de tasas de baudios [24]

Entrada/Salida paralelo (PIO)

El puerto de E/S paralelo tiene dos registros de 32bit dedicados al control de los pines del puerto. Hay 4 registros de tamaño byte para la salida del puerto y otros 4 registros tamaño byte para la entrada del mismo. Esos dos conjuntos de 32bit cada uno se ubican en la misma dirección de memoria. Si se lee de cada uno de los registros, se estaría accediendo a la entrada del puerto, y si se escribe en ellos, se accedería a la salida del puerto. Al igual que pasaba con la UART, los registros se encuentran ubicados en direcciones consecutivas de memoria, permitiendo el acceso de operandos tamaño byte, word o long. Las salidas del puerto son inicializadas al valor 0x00 cuando se aplica un reset.

Las señales de E/S del puerto paralelo:

Clase	Nombre de Señal	Dirección	Significado	Notas
Señales del Sistema	CLK	Entrada	Reloj del sistema	
	RST	Entrada	Reset	
Señales del Bus WISHBONE	CE	Entrada	Chip Select (Selección de Módulo)	STB
	WE	Entrada	Write Enable	
	SEL[3:0]	Entrada	Selección de byte	
	DATI[31:0]	Entrada	Dato de entrada (Dato escrito por CPU)	
	DATO[31:0]	Salida	Dato de salida (Dato leído)	
Puerto paralelo	PI[31:0]	Entrada	Entrada puerto	
	PO[31:0]	Salida	Salida puerto	

Tabla 3.7 Señales de E/S del puerto paralelo (PIO) [24]

Los registros de salida:

[PORT Output] Address=0xABCD0000 W only reserved

31(7)	30(6)	29(5)	28(4)	27(3)	26(2)	25(1)	24(0)
reserved	Reserved	reserved	reserved	reserved	reserved	Reserved	reserved

[PORT Output] Address=0xABCD0001 W only KEYYO(KEY Matrix Y-axis Output)

23(7)	22(6)	21(5)	20(4)	19(3)	18(2)	17(1)	16(0)
reserved	Reserved	reserved	KY4	KY3	KY2	KY1	KY0

[PORT Output] Address=0xABCD0002 W only LCDCON (LCD Control Output)

15(7)	14(6)	13(5)	12(4)	11(3)	10(2)	9(1)	8(0)
reserved	Reserved	reserved	reserved	reserved	E	R/W	RS

[PORT Output] Address=0xABCD0003 W only LCDOUT (LCD Write Data Output)

7(7)	6(6)	5(5)	4(4)	3(3)	2(2)	1(1)	0(0)
DW7	DW6	DW5	DW4	DW3	DW2	DW1	DW0

Los registros de entrada:

[PORT Input] Address=0xABCD0000 R only reserved

31(7)	30(6)	29(5)	28(4)	27(3)	26(2)	25(1)	24(0)
reserved	Reserved	reserved	reserved	reserved	reserved	Reserved	reserved

[PORT Input] Address=0xABCD0001 R only KEYXI (KEY Matrix X-axis Input)

23(7)	22(6)	21(5)	20(4)	19(3)	18(2)	17(1)	16(0)
reserved	Reserved	reserved	KX4	KX3	KX2	KX1	KX0

[PORT Input] Address=0xABCD0002 R only reserved

15(7)	14(6)	13(5)	12(4)	11(3)	10(2)	9(1)	8(0)
reserved	Reserved	reserved	reserved	reserved	E	R/W	RS

[PORT Input] Address=0xABCD0003 R only LCDIN (LCD Read Data Input)

7(7)	6(6)	5(5)	4(4)	3(3)	2(2)	1(1)	0(0)
DR7	DR6	DR5	DR4	DR3	DR2	DR1	DR0

Tabla 3.8 Registros del puerto paralelo de entrada / salida [24]

Describimos los registros de la figura anterior:

- **1º BYTE**: Reservado.
- **KEYYO/KEYXI**: Registro dedicado a controlar el teclado incluido en la placa donde Aquarius estaba implementado originariamente.
- **LCDCON**: Registro destinado al control de la pantalla LCD contenida en la placa en la que Aquarius estaba implementado originariamente. Sólo útil en escritura, reservado en lectura.
- **LCDOUT/LCDIN**: Registro en el que se almacena el dato que se envía a la pantalla LCD, en el caso de escritura, que procede del puerto paralelo en caso de lectura.

El fichero Verilog correspondiente al puerto paralelo es *pio.v*.

Controlador del sistema (System Controller)

El controlador del sistema posee las siguientes funciones:

1. Generar la excepción de un evento hardware.
   - NMI (ruptura de dirección (Address Break))
   - IRQ (debida a un intervalo del Timer)
   - Error de dirección CPU (se vigilan las transacciones WISHBONE)
2. Emular la excepción de un evento hardware.
   - NMI (Non-Maskable-Interrupt: Interrupción no enmascarable)
   - IRQ
   - Error de dirección CPU
   - Error de dirección DMA
   - Reset manual
3. Control del nivel de prioridad entre las distintas peticiones de excepción hardware.
4. Configuración de los niveles de prioridad IRQ y del número de vector.
5. Intervalo de 12bit del Timer para la generación de la IRQ.
6. Función de ruptura del bus de dirección (Address Bus Break Function) para la capacidad de depurado (NMI).
7. Detección del error de dirección CPU vigilando las señales del bus WISHBONE.
8. Control de baja potencia y modo SLEEP.

Las señales de E/S del controlador del sistema se muestran a continuación:

Clase	Nombre de Señal	Dirección	Significado	Notas
Señales del sistema	CLK_SRC	Entrada	Reloj del sistema fuente	
	CLK	Salida	CLK , que se para en SLEEP	
	SLP	Entrada	Petición SLEEP desde CPU	
	WAKEUP	Entrada	Petición de despertar	
	RST	Entrada	Reset	
Señales del Bus WISHBONE	CE	Entrada	Chip Select (Selección de módulo)	STB
	WE	Entrada	Write Enable	
	SEL[3:0]	Entrada	Selección de byte	
	ACK	Entrada	Asentimiento de Bus	
	DATI[31:0]	Entrada	Dato Entrada (Dato escrito por CPU)	
	DATO[31:0]	Salida	Dato Salida (Dato leído)	
	STB	Entrada	Strobe (El Bus monitoriza BRK)	
	ADR[31:0]	Entrada	Dirección (El Bus monitoriza BRK)	
Eventos Hardware	EVENT_REQ[2:0]	Salida	Petición de Evento	
	EVENT_INFO[11:0]	Salida	Información de Evento (IRQ)	
	EVENT_ACK	Entrada	Asentimiento de Evento desde CPU	

Tabla 3.9 Señales de E/S del controlador del sistema [24]

El controlador del sistema posee dos registros de 32bit; INTCTL y BRKADR. Ambos registros sólo pueden ser accedidos por un operando long, es decir, de 32bit. Los valores iniciales que toman estos registros cuando tiene lugar un reset son: 0x00000FFF para INTCTL, y 0x00000000 para BRKADR. Los registros correspondientes al controlador del sistema se muestran a continuación:

[SYS] Address=0xABCD0200 R/W INTCON (Interrupt Control)

31	30	29	28	27	26	25	24
E_NMI	E_IRQ	E_CER	E_DER	E_MRS	reserved	TMRON	BRKON
23	22	21	20	19	18	17	16
ILVL3	ILVL2	ILVL1	ILVL0	IVEC7	IVEC6	IVEC5	IVEC4
15	14	13	12	11	10	9	8
IVEC3	IVEC2	IVEC1	IVEC0	TMR11	TMR10	TMR9	TMR8
7	6	5	4	3	2	1	0
TMR7	TMR6	TMR5	TMR4	TMR3	TMR2	TMR1	TMR0

[SYS] Address=0xABCD0204 R/W BRKADR (Break Address)

31	30	29	28	27	26	25	24
ADR31	ADR30	ADR29	ADR28	ADR27	ADR26	ADR25	ADR24
23	22	21	20	19	18	17	16
ADR23	ADR22	ADR21	ADR20	ADR19	ADR18	ADR17	ADR16
15	14	13	12	11	10	9	8
ADR15	ADR14	ADR13	ADR12	ADR11	ADR10	ADR9	ADR8
7	6	5	4	3	2	1	0
ADR7	ADR6	ADR5	ADR4	ADR3	ADR2	ADR1	ADR0

Tabla 3.10 Registros del controlador del sistema [24]

Describimos los registros:

- **INTCON**: Registro destinado al control de las interrupciones del sistema.
- **BRKADR**: Registro dedicado a la interrupción no enmascarable, NMI.

El fichero Verilog correspondiente a este módulo en *sys.v*.

Top

Es interesante ver las entradas y salidas del módulo Verilog *top.v*, el módulo colocado más arriba en cuanto a jerarquía, ya que nos permite entender mejor a qué entradas y salidas del módulo *top.v* se conectan las entradas y salidas de cada uno de los periféricos internos, así como de la CPU. Se muestran a continuación:

Clase	Nombre de señal	Dirección	Significado	Notas
Señales del Sistema	CLK_SRC	Entrada	Reloj del sistema	
	RST_n	Entrada	Reset	Negated
Puerto Paralelo de E/S	LCDRS	Salida	Selección de registro LCD	PO[8]
	LCDRW	Salida	Lectura/Escritura LCD	PO[9]
	LCDE	Salida	Señal habilitación LCD	PO[10]
	LCDDBO[7:0]	Salida	Bus Datos de Salida LCD	PO[7:0]
	LCDDBI[7:0]	Entrada	Bus Datos de Entrada LCD	PI[7:0]
	KEYYO[4:0]	Salida	Salida KEY Matriz Y	PO[20:16]
	KEYXI[4:0]	Entrada	Entrada KEY Matriz X	PI[20:16]
UART	RXD	Entrada	Dato Serie Rxón	
	TXD	Salida	Dato Serie Txón	
	CTS	Entrada	Listo para Envío	
	RTS	Salida	Petición de Envío	

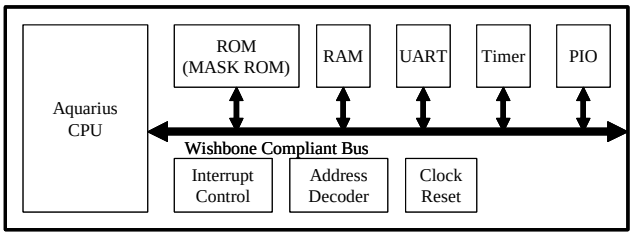
Tabla 3.11 Señales de E/S del top [24]

3.2.1.d Ejemplos de configuraciones usando Aquarius

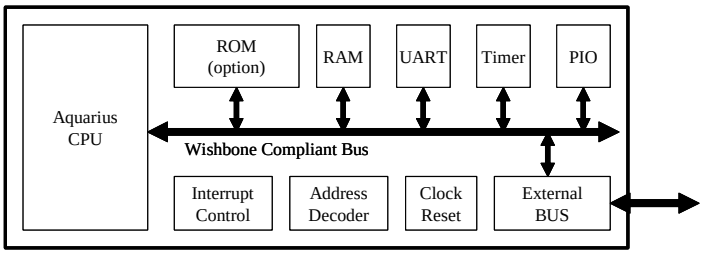
La figura 3.9 muestra diversas configuraciones que son posibles usando Aquarius.

- Vemos que puede usarse como microcontrolador (marcada como (1) en la figura 3.9),
- También como microcontrolador con bus externo para conexión con memorias externas, por ejemplo (marcada como (2)),
- Como microcontrolador de alta velocidad con bus externo, dividiendo en este caso el bus interno en dos, uno de alta velocidad para la CPU y memorias rápidas, y otro de baja velocidad, para periféricos y las interfaces de memoria externa (marcada como (3)).

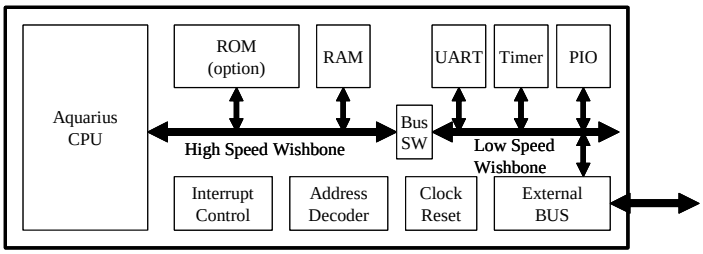
Este hecho no hace sino mostrar la gran flexibilidad que poseen los sistemas en-chip que usan Aquarius.



(1) Micro Controller using Aquarius



(2) Micro Controller using Aquarius with external bus



(3) High Speed Micro Controller using Aquarius with external bus

Figura 3.9 Ejemplos de SoC basados en Aquarius [24]

3.2.2 Programación de aplicaciones

En este apartado detallaremos las herramientas y los pasos necesarios para pasar un programa de aplicación, escrito en lenguaje de alto nivel, a un formato adecuado para su simulación a nivel hardware y su posterior implementación. Todo ello usando herramientas de libre distribución, sin coste económico alguno por parte del usuario.

Para explotar la funcionalidad del procesador que estamos estudiando, hemos de escribir programas en su memoria interna, siguiendo un flujo de aplicación que parte de un programa en lenguaje C, de alto nivel, y que concluirá con la síntesis de un fichero Verilog, que describirá el contenido de la memoria interna del microprocesador que hace que éste ejecute el programa deseado. El diagrama se muestra a continuación:

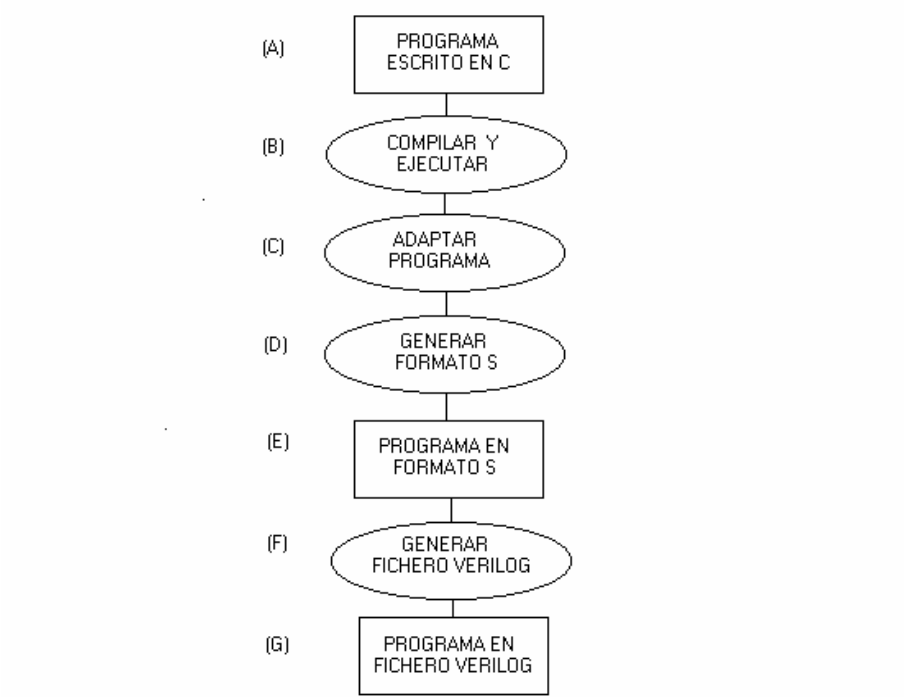


Figura 3.10 Diagrama de aplicación

Para ello usaremos unas herramientas concretas, accesibles a cualquier usuario y libres, que permitirán la ejecución de las instrucciones que usa el microprocesador Aquarius. El juego de instrucciones es SuperH-2 ISA (Instruction Set Architecture), de 16 bits que hacen que el hardware sea sencillo y que el código objeto compilado, procedente de programas fuente escritos en C, tenga un tamaño reducido.

Para poder trabajar con el procesador, debemos descargar cierto material de la página *http://www.opencores.org/cvsweb.shtml/Aquarius*, en la que nos encontraremos cinco directorios cuyo contenido comentamos a continuación.

- El directorio *doc* contiene una descripción del sistema,
- *verilog* contiene los ficheros Verilog necesarios para implementar el procesador,
- *tools* guarda una serie de herramientas útiles, como *genrom.c*, cuya utilidad comentaremos más adelante,
- *fpga* tiene otras utilidades como *genram.c*, que también nombraremos a continuación,
- *application* contiene ejemplos de programas en código C, acompañados de una serie de ficheros necesarios para la implementación, tales como el *makefile* o el fichero de cabecera *common.h*, que deben ser utilizados por los usuarios que deseen programar sus propias aplicaciones en el sistema.

Herramientas necesarias e instalación de las mismas

Sería bueno integrar todas las herramientas en un entorno único de trabajo. El problema es que el compilador y ensamblador de SuperH-2 trabajan en UNIX y la mayoría de herramientas de desarrollo para FPGA lo hacen en WINDOWS. La solución pasa por usar CYGWIN, que hace funcionar UNIX en un entorno

WINDOWS. Podemos descargarnos CYGWIN desde la página http://www.cygwin.com , procurando que la descarga sea lo más completa posible. Además, habría que añadir unas librerías que hagan que el ensamblador generado al compilar sea el deseado. Las librerías que debemos descargar son las que siguen:

- Descargar de la página ftp://ftp.gnu.org/pub/gnu/ :

  binutils-2.13.1.tar.gz
  gcc-2.95.3.tar.gz
  gdb-5.2.1.tar.gz

- Descargar de la página http://sources.redhat.com/newlib/ :

  newlib-1.10.0.tar.gz

Para instalar las librerías descargadas debemos realizar una serie de pasos que detallamos a continuación:

- Guardar las librerías anteriores bajo /usr/local/src.
- Instalamos GNU binutils.

```
cd /usr/local/src
gzip –dc binutils-2.13.1.tar.gz | tar xvf –
cd binutils-2.13.1
mkdir work
cd work
../configure –prefix=/usr/local –target=sh-elf
make
make install
```

- Instalamos GNU gcc y newlib:

```
cd /usr/local/src
gzip –dc newlib-1.10.0.tar.gz | tar xvf –
gzip –dc gcc-2.95.3.tar.gz | tar xvf –
cd gcc-2.95.3
ln –s ../newlib-1.10.0/newlib .
mkdir work
cd work
../configure --prefix=/usr/local --target=sh-elf --with-gnu-as --with-gnu-ld --with-dwarf2 --disable-multilib --enable-languages=c --with-newlib
make
make install
```

- Instalamos GNU gdb:

```
cd /usr/local/src
gzip –dc gdb-5.2.1.tar.gz | tar xvf –
cd gdb-5.2.1
mkdir work
cd work
```

```
../configure  –prefix=/usr/local –target=sh-elf
make
make  install
```

NOTA: Dependiendo del procesador del ordenador en que se instale, cada una de las tres últimas instrucciones de cada grupo, es decir, *configure*, *make* y *make install*, pueden requerir bastante tiempo para su ejecución.

Las herramientas de simulación y síntesis que usaremos son ISE 8.1 y MODELSIM XE 5.5e (Starter Edition).

Desarrollo de aplicaciones

Una vez indicadas las herramientas necesarias vamos a explicar el diagrama de aplicaciones de la figura 3.10. Para entenderlo mejor, veamos el flujo mediante un ejemplo simple. Nos encontramos en el paso (A) del diagrama. Supongamos que nuestro programa de aplicación es el siguiente:

```
     /*Programa que consiste en incrementar una variable, x,
hasta alcanzar el valor dado por M, en este caso 100*/

 #include <stdio.h>
  #define M 100

  int main (void)
  {
     int x=1;

     while(x<M)
     {
         x=x+1;
     }

     printf (“\n El resultado final es %d \n”,x);
     return 0;
   }
```

El siguiente paso es compilar en Cygwin el programa y verificar su correcto funcionamiento. Se trata del paso (B):

Figura 3.11 Resultado tras compilar programa.

Ahora nos encontramos en (C). Hay que adaptar el código a las librerías a las que hemos hecho referencia anteriormente:

```
#include <stdio.h>
#include “common.h”
#define M 100

int main_sh (void)              /*hay que añadir _sh*/
{
   int x=1;

   while(x<M)
   {
       x=x+1;
   }

 /*eliminamos funciones que no tienen reflejo en instrucciones
SuperH, como printf*/

      return 0;
  }
```

Ahora hay que configurar que al compilar la aplicación obtengamos un ensamblador en formato SuperH-2, para lo cual, configuramos un fichero *makefile* de tal modo que, al lanzar la orden *make*, dentro de la consola CYGWIN, obtengamos el fichero en formato S. Para ello hay que indicar, mediante una carpeta contenida en la de aplicación de nombre *startup*, y mediante dos ficheros, *crt0.s* y *sh.x* en qué parte del mapa de memoria vamos a almacenar el programa generado. En este caso se trata de la rom interna como vemos en el fichero sh.x del ejemplo:

```
                ...

        OUTPUT_FORMAT("elf32-sh")
        OUTPUT_ARCH(sh)

        MEMORY
        {
          rom    : o = 0x00000000, l = 0x1e00
          stack  : o = 0x00001e00, l = 0x0200
        }

        SECTIONS
        {
         .text :      {
                        *(.text)
                        *(.strings)
                        _etext = . ;
                      }  > rom
                ...
```

De este modo, compilamos y generamos el fichero en formato S, paso (D) del diagrama, simplemente ejecutando la orden *make*:

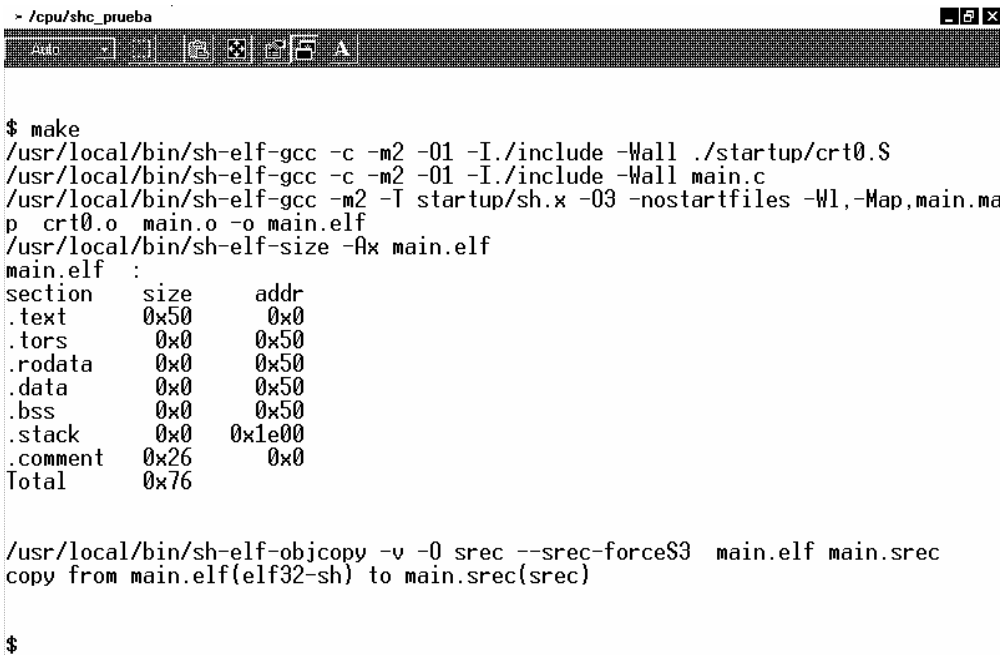


Figura 3.12 Resultado de ejecutar la orden make.

Ahora tenemos ya generado el fichero en formato S, *main.srec*. Nos encontramos en el paso (E). Este fichero debe ser pasado por una aplicación que genere el fichero verilog que pasaremos al simulador. Para ello disponemos de dos herramientas más, dos programas en lenguaje C llamados *genrom.c* y *genram.c*, que debemos compilar en nuestra consola CYGWIN para que nos aparezca el ejecutable. Ambos programas hacen lo mismo, cambiando únicamente el formato final del fichero obtenido.

Genrom.c genera *rom.v*, que guarda el contenido de la memoria interna del microprocesador y que será el fichero que usemos para la simulación.

Genram.c genera *ram.dat*, que es otra forma de configurar la ROM, más adecuada para la implementación en FPGA, pues lo hace mediante BlockRAM, reduciendo considerablemente el consumo de puertas lógicas.

Vamos a generar *rom.v* (paso (F)) para realizar una simulación funcional, para ello, una vez tengamos el ejecutable de *genrom.c*, *genrom.exe*, lo invocamos pasando como parámetro el fichero en formato S, *main.srec*, como vemos a continuación:

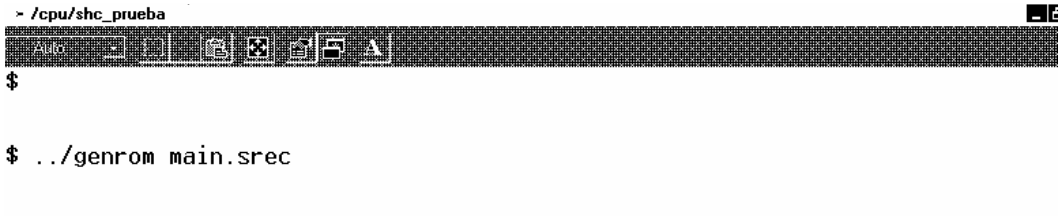


Figura 3.13 Resultado de ejecutar el programa genrom.exe

Ya tenemos el fichero Verilog, *rom.v*, (paso (G)).Vemos el contenido de *rom.v*:

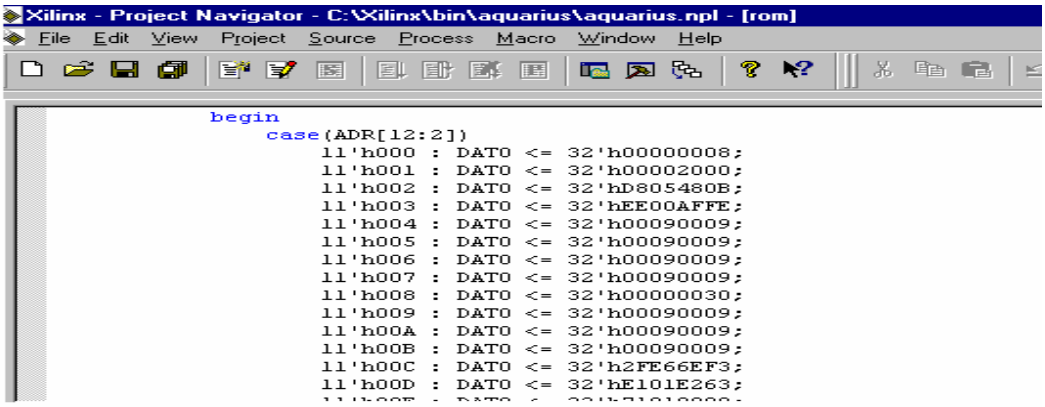


Figura 3.14 Contenido del fichero *rom.v*

Una vez tenemos el fichero *rom.v*, arrancamos el programa ISE y creamos un nuevo proyecto, en el que cargaremos los ficheros Verilog correspondientes al procesador Aquarius. Configuramos el proyecto para ser trasvasado a una FPGA Virtex E (XCV300E-8PQ240) y, una vez cargado el proyecto, creamos un fichero de prueba que sencillamente resetee el sistema.

Lanzamos el simulador *Modelsim* y obtenemos los siguientes resultados:

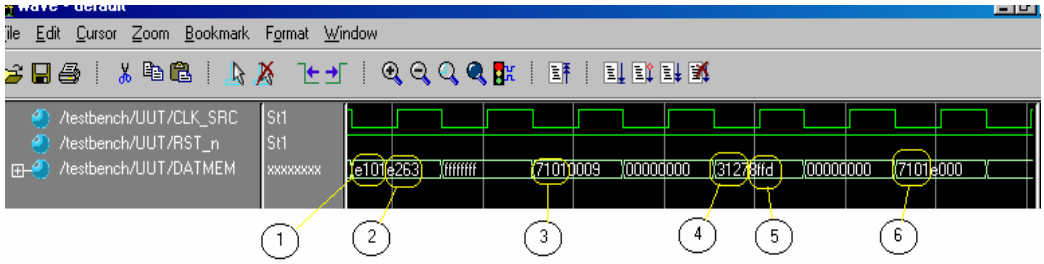


Figura 3.15 Pantalla de simulación del programa en plataforma Modelsim XE 5.5e

Observamos cómo va ejecutando las instrucciones almacenadas en memoria, y si vemos a qué código corresponde cada una, descubriremos que opera correctamente. Hay que tener en cuenta que el tamaño del bus es de 32 bits, por lo que en cada fase de recogida de instrucciones nos haremos con dos cada vez, ya que éstas son de 16 bits. La figura anterior muestra las instrucciones correspondientes a la función *main* del código C mostrado al inicio del apartado. Basándonos en los códigos de instrucciones del SuperH-2 y siguiendo la numeración de la figura 3.15:

Nº	Código	Traducción	Comentario
1	E101	Mov #1,R1	Almacena 1 en R1 (x)
2	E263	Mov #63,R2	Guarda 99 (63 en hexadecimal) en R2 (M)
3, 6	7101	Add #1,R1	Suma una unidad a R1 (x++)
4	3127	Cmp/gt R2,R1	Compara si R1 es mayor que R2
5	8FFD	Bf/s FD	Si es falso, repito el bucle

Tabla 3.11 Traducción de los códigos de instrucción

Vemos cómo vuelve a repetirse el bucle, ya que las instrucciones mostradas son las iniciales, con valores de la variable *x* aún pequeños. Hay instrucciones que hemos obviado al no tener una correspondencia directa con el código C, como la preparación del salto a la rutina *main*, el salvado en pila de variables, etc., previas a las instrucciones indicadas anteriormente.

3.2.3 Necesidad de memorias de puertos múltiples

En este punto vamos a comentar por qué razón debemos añadir más memoria al sistema y las causas por las que esa memoria debe ser de puertos múltiples.

En nuestro sistema de visión trabajaremos con imágenes de tamaño reducido para simplificar las simulaciones principalmente, aunque puede funcionar con imágenes de cualquier tamaño. Sólo habría que modificar ese parámetro en unas líneas del código fuente del sistema coprocesador, nunca del procesador Aquarius, lo cual resulta bastante sencillo. Pese a la escasa dimensión de las imágenes que vamos a manejar, bastaría tener una imagen de 64x64, que es un tamaño reducido, para rellenar la mitad de la RAM interna de Aquarius. Es por ello que requerimos un suplemento de memoria RAM.

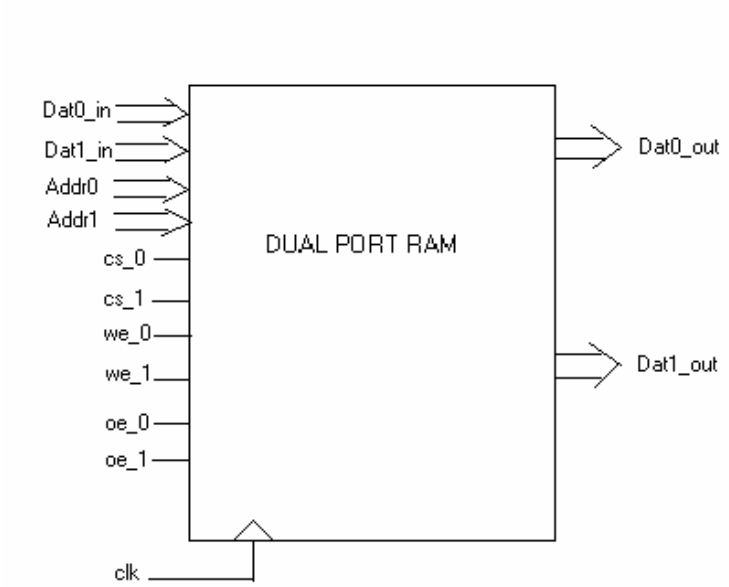


Figura 3.16 Memoria RAM de doble puerto

Por otra parte, nuestro sistema coprocesador trabajará de forma independiente al procesador Aquarius, pese a que va a procesar un objeto común, la imagen en cuestión. La única solución viable es una memoria de doble puerto, que impida el acceso simultáneo, al menos en escritura, a través de sus dos puertos. Esto permitiría al procesador Aquarius almacenar la imagen recibida por el puerto serie en la memoria de doble puerto, avisar al coprocesador que procese la imagen y desentenderse del proceso. Todo ello sin peligro de cortocircuito, ya que el coprocesador accederá a la imagen por el segundo puerto de la memoria RAM de doble puerto.

3.3 Coprocesador visual

3.3.1 Estructura del coprocesador

La figura 3.17 muestra la estructura interna del coprocesador:

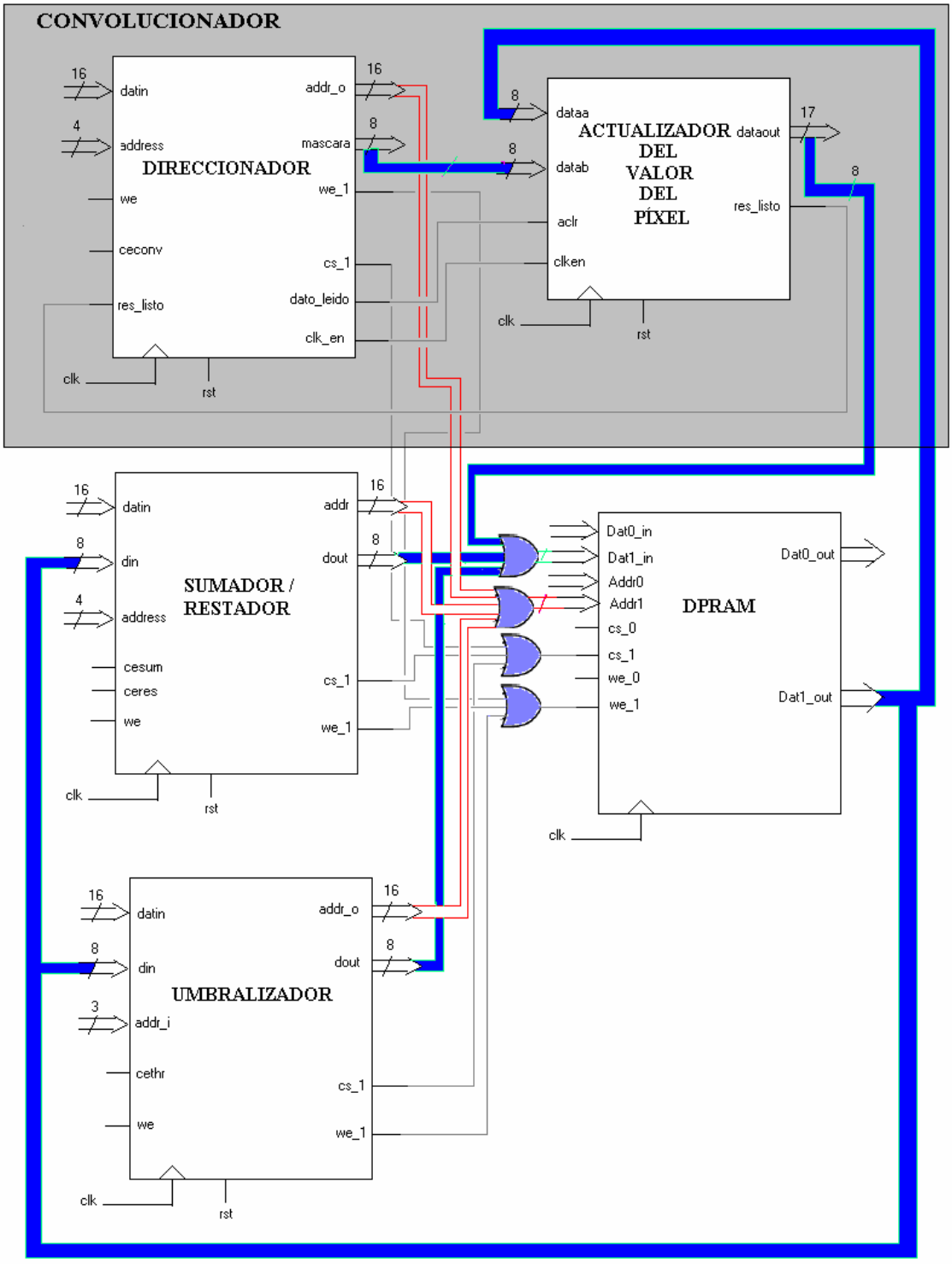


Tabla 3.17. Esquema interno del coprocesador

El coprocesador visual es una entidad encargada de procesar imágenes mediante el paso de una máscara que actúa sobre los píxeles de las mismas. En

el posible esquema que hemos pensado para experimentar con el sistema de visión, mostrado en la figura 3.1, estas imágenes son recibidas por Aquarius a través de la UART (puerto serie), y posteriormente almacenadas en un dispositivo de memoria de doble puerto, desde donde son leídas por el coprocesador. Además, tanto éste como la DPRAM deben cumplir con la interfaz WISHBONE, ya que es la usada por Aquarius, al que se encuentran conectados, como se muestra en la siguiente figura:

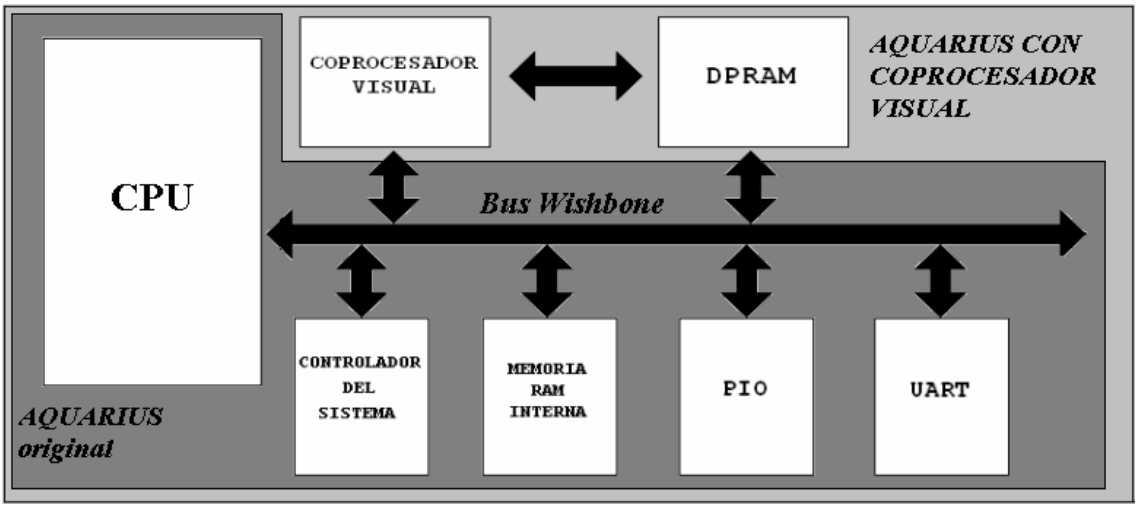


Figura 3.18. Estructura del sistema de visión.

El coprocesador está formado por varios subsistemas. Éstos sirven para procesar una imagen que está almacenada dentro de una memoria de doble puerto, también situada dentro del coprocesador. Los subsistemas que forman parte del coprocesador son:

- *Convolucionador*: Encargado de realizar el paso de la máscara de procesado sobre una imagen almacenada en memoria.
- *Sumador / Restador de imágenes*: Bloque encargado de realizar una suma o una resta de dos imágenes almacenadas en memoria.
- *Umbralizador*: Lleva a cabo un umbralizado de una imagen almacenada en memoria.

Todos estos bloques internos del coprocesador pueden verse con detalle en la figura 3.17. En ella, todas las señales no conectadas se corresponden con puertos de entrada/salida de la entidad coprocesador (ver figura 3.19). Las líneas de ancho mayor de 1bit están representadas por flechas gruesas, siendo de color azul las líneas de datos y de color rojo las de direcciones. También aparecen unas puertas lógicas de tipo OR, que darán el control del puerto secundario de la memoria de doble puerto a quien active sus líneas. Como es el procesador quien manda activarse a los distintos subsistemas internos al coprocesador, no existe peligro de conflicto por intentar forzar dos valores lógicos distintos en esos puntos. La siguiente figura muestra el coprocesador visto como una caja negra:

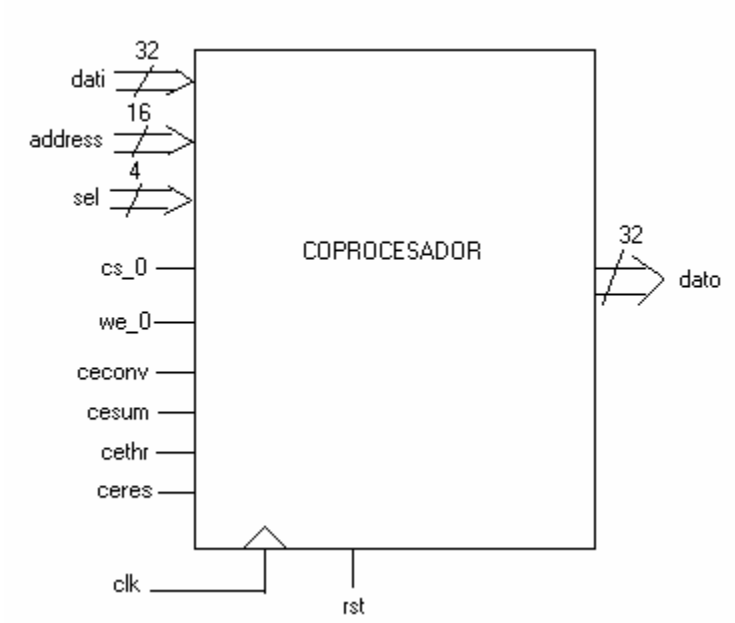


Figura 3.19. Interfaz de E/S del coprocesador

En la figura, aparecen varias señales que explicamos a continuación:

- **CLK**: Reloj del sistema.
- **RST**: Señal de reset del sistema.
- **DATI [31:0]**: Bus de datos de 32bit procedente de Aquarius. Es una señal de entrada al dispositivo.
- **ADDRESS [15:0]**: Bus de direcciones de 16bit. Se corresponden con los 16 bits menos significativos de la línea de direcciones de Aquarius (su bus de direcciones es de 32bit). Es una señal de entrada.
- **CS_0**: Línea de habilitación de la memoria de doble puerto (DPRAM: Dual Port RAM). Procedente de Aquarius, es de entrada al coprocesador.
- **WE_0**: Señal de escritura. Se corresponde con la señal WE de Aquarius. Cuando está activa indica que el ciclo actual es de escritura. Es una señal de entrada.
- **CECONV**: Señal de habilitación del convolucionador. Útil para que Aquarius escriba en los registros internos de la unidad de convolución que se halla dentro del coprocesador. Es una señal de entrada.
- **CESUM**: Señal de habilitación del sumador. Sirve para que Aquarius escriba en los registros internos del sumador del coprocesador. Es una señal de entrada.
- **CETHR**: Señal de habilitación del umbralizador. Útil para que Aquarius escriba en los registros internos de la unidad de umbralizado que se encuentra dentro del coprocesador. Es una señal de entrada.
- **CERES**: Señal de habilitación del restador. Sirve para que Aquarius escriba en los registros internos del restador del coprocesador. Es una señal de entrada.
- **DATO [31:0]**: Bus de datos de 32bit de salida. Contendrá los datos que el coprocesador vuelca al bus de datos de Aquarius. Es, por tanto una señal de salida.

Antes de detallar a fondo cada uno de los subsistemas que aparecen en la figura 3.17, vamos a detallar cómo es la conexión física entre Aquarius y el coprocesador.

3.3.2 Interfaz Aquarius – coprocesador visual

Vamos a ver cómo llevamos a cabo la tarea de integración del coprocesador dentro de la arquitectura del procesador, para que éste pueda acceder a los distintos subsistemas del coprocesador. Este proceso se realiza a nivel hardware y a nivel software.

Conexión del hardware

Para realizar la conexión física entre el procesador, Aquarius, y el coprocesador visual diseñado es necesario realizar unos ajustes en Aquarius, para que su funcionamiento normal no se vea afectado negativamente y para que pueda acoger al nuevo sistema y acceder sin problemas a sus puertos. Para ello, lo primero que hay que hacer es incluir a los subsistemas internos del coprocesador en el mapa de memoria de Aquarius, para que éste pueda acceder a ellos. Hay que recalcar en este punto que es Aquarius el que accede directamente a los subsistemas del coprocesador, por lo que debe incluir a cada uno de ellos en el mapa de memoria, no al coprocesador en conjunto.

El siguiente paso es ver es qué direcciones de memoria van a ocupar cada uno de los subsistemas. Para ello observamos el mapa de memoria original de Aquarius, que aparece en la tabla 3.2. Es importante observar en el mapa de memoria original que dependiendo del valor que tomen determinadas líneas de dirección varíe el número de ciclos de reloj que tarda en completarse un acceso a una dirección de memoria. Observamos que los periféricos originales de Aquarius están situados en unas direcciones de memoria en las que cada acceso en lectura / escritura conlleva cuatro ciclos de reloj y que la memoria interna del procesador lo está en otras en las que cada acceso dura un ciclo de reloj. Es lógico, por tanto, pensar que debemos colocar nuestros subsistemas en zonas con accesos de 4 ciclos de reloj. La condición que debe darse para ello es que el decimoséptimo bit del bus de direcciones valga 1, es decir, ADR[16]=1.

Se han realizado pruebas colocándolos en zonas de acceso de 1 ciclo de duración y se ha comprobado cómo no le da tiempo de reaccionar al subsistema probado. Aplicamos la misma idea para la memoria de doble puerto. También se ha observado que era importante ubicar los nuevos periféricos que va a tener Aquarius en direcciones de memoria que permitan accesos de 32bit, para que sean tratados como los periféricos originales del procesador, para lo que era necesario que el decimoctavo bit del bus de direcciones tuviera el valor 0, es decir, ADR[17]=0.

Así, el nuevo mapa de memoria de Aquarius quedaría:

Address	Device	Size	Access Cycle	IF Width	Notes
0x00000000-0x00001FFF	ROM	8KB	1cyc	32bit	A
0x00002000-0x00003FFF	RAM	8KB	1cyc	32bit	B
0x00004000-0x0000FFFF	Shadow of 0x00000000-0x00003FFF				
0x00010000-0x00011FFF	ROM	8KB	4cyc	32bit	Shadow of A
0x00012000-0x00013FFF	RAM	8KB	4cyc	32bit	Shadow of B
0x00014000-0x0001FFFF	Shadow of 0x00010000-0x00013FFF				
0x00020000-0x00021FFF	ROM	8KB	1cyc	16bit	Shadow of A
0x00022000-0x00023FFF	RAM	8KB	1cyc	16bit	Shadow of B
0x00024000-0x0002FFFF	Shadow of 0x00020000-0x00023FFF				
0x00030000-0x00031FFF	ROM	8KB	4cyc	16bit	Shadow of A
0x00032000-0x00033FFF	RAM	8KB	4cyc	16bit	Shadow of B
0x00034000-0x0003FFFF	Shadow of 0x00030000-0x00033FFF				
0x00040000-0xABCCFFFF	Shadow of 0x00000000-0x0003FFFF				
0xABCD0000-0xABCD00FF	PIO	256B	4cyc	32bit	
0xABCD0100-0xABCD01FF	UART	256B	4cyc	32bit	
0xABCD0200-0xABCD02FF	SYS	256B	4cyc	32bit	
0xABCD0300-0xABCD03FF	CONV	256B	4cyc	32bit	
0xABCD0400-0xABCD04FF	SUM	256B	4cyc	32bit	
0xABCD0500-0xABCD05FF	THR	256B	4cyc	32bit	
0xABCD0600-0xABCD06FF	RES	256B	4cyc	32bit	
0xABCD0700-0xFFFCFFFF	Shadow of 0x00000000-0x0003FFFF				
0xFFFD0000-0xFFFDFFFF	DPRAM	64KB	4cyc	32bit	
0xFFFE0000-0xFFFFFFFF	Shadow of 0x00000000-0x0003FFFF				

Tabla 3.12. Mapa de memoria modificado de Aquarius.

Hay que hacer notar que este es un posible mapa de memoria, ya que existen otras muchas combinaciones posibles que dan lugar a mapas de memoria alternativos y perfectamente válidos. Otra tarea a realizar es modificar el fichero Verilog de más alto nivel de Aquarius, *top.v*, para que seleccione los subsistemas añadidos al mapa de memoria cuando éstos sean direccionados.

La siguiente figura muestra un extracto del contenido del fichero *top.v* original, es decir, que sólo debía acceder a memoria interna, E/S paralelo, UART, y al controlador del sistema:

```
                    ...

always @(DATMEM or DATPIO or DATUART or DATSYS) begin
     DATI <= DATMEM | DATPIO | DATUART | DATSYS; // read data gathering
  end

always @(STB or ADR)
 begin
    if (STB == 1'b0)
       {CEMEM,CEPIO,CEUART,CESYS} <= 4'b0000;
    else if (ADR[31:8] == 24'hABCD00)
       {CEMEM,CEPIO,CEUART,CESYS} <= 4'b0100;
    else if (ADR[31:8] == 24'hABCD01)
       {CEMEM,CEPIO,CEUART,CESYS} <= 4'b0010;
    else if (ADR[31:8] == 24'hABCD02)
       {CEMEM,CEPIO,CEUART,CESYS} <= 4'b0001;
    else
       {CEMEM,CEPIO,CEUART,CESYS} <= 4'b1000;
 end
                    ...
```

En este fragmento de código pueden observarse varias cosas:

- En el bloque *always* que aparece en la parte superior se observa cómo todos los periféricos de Aquarius vuelcan sus datos en una misma señal, llamada *DATI*, a través de una operación del tipo OR. Esto es lo mismo que se realizó con los subsistemas del coprocesador al acceder a la memoria de doble puerto.
- En el segundo bloque de este tipo vemos la lógica de selección de dispositivos. Aparecen las señales STB y ADR. La primera nos indica que hay un ciclo de bus válido y la segunda nos dice qué dispositivo se encuentra implicado en él. Observamos la correspondencia existente con el mapa de memoria original de Aquarius expuesto en la tabla 3.2.

Ahora vamos a mostrar el fichero *top.v* tras la remodelación sufrida para hacer posible la integración del coprocesador. Para ello vamos a ver cómo se ve afectada la misma porción de código anterior:

```
                    ...
always @(DATMEM or DATPIO or DATUART or DATSYS or DATCOP) begin
     DATI <= DATMEM | DATPIO | DATUART | DATSYS | DATCOP; // read data gathering
  end

  always @(STB or ADR)
  begin
     if (STB == 1'b0)
        {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b000000000;
     else if (ADR[31:8] == 24'hABCD00)
        {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b010000000;
     else if (ADR[31:8] == 24'hABCD01)
        {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b001000000;
     else if (ADR[31:8] == 24'hABCD02)
        {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b000100000;
     else if (ADR[31:8] == 24'hABCD03)
        {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b000010000;
     else if (ADR[31:8] == 24'hABCD04)
        {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b000001000;
     else if (ADR[31:8] == 24'hABCD05)
```

```
        {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b000000100;
    else if (ADR[31:8] == 24'hABCD06)
        {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b000000010;
    else if (ADR[31:16] == 16'hFFFD)
        {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b000000001;
    else
        {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b100000000;
end
```

...

En este nuevo fichero se observa cómo introducimos los nuevos periféricos a la arquitectura de Aquarius, respetando la filosofía usada para acceder a los periféricos originales del procesador. Un detalle a destacar es que, tanto en el fichero original de Aquarius como en el modificado, el acceso por defecto se realiza a memoria interna, es decir, si accedo a una dirección que no corresponde a ninguno de los periféricos integrados en el mapa de memoria del sistema, la señal de habilitación que se activa es la correspondiente a memoria interna, es decir, DATMEM. Por supuesto, hay que añadir el módulo correspondiente al coprocesador, llamado *toptotal*, al fichero *top.v*, conectando adecuadamente sus entradas y salidas, así como definir las variables necesarias para ello.

...

```
//*************
// Modules
//*************
    cpu CPU(
        // system signal
        .CLK(CLK), .RST(RST),
        // WISHBONE external bus signal
        .CYC_O(CYC), .STB_O(STB), .ACK_I(ACK),
        .ADR_O(ADR), .DAT_I(DATI), .DAT_O(DATO),
        .WE_O(WE), .SEL_O(SEL),
        .TAG0_I(IF_WIDTH),
        // Exception
        .EVENT_REQ_I(EVENT_REQ),
        .EVENT_ACK_O(EVENT_ACK),
        .EVENT_INFO_I(EVENT_INFO),
        //SLEEP
        .SLP_O(SLP)
    );

    memory MEMORY(
        .CLK(CLK), .CE(CEMEM), .WE(WE), .SEL(SEL),
        .ADR(ADR[13:0]), .DATI(DATO), .DATO(DATMEM)
    );

    pio PIO(
        .CLK(CLK), .RST(RST),
        .CE(CEPIO), .WE(WE), .SEL(SEL),
        .DATI(DATO), .DATO(DATPIO),
        .PI(PI), .PO(PO)
    );

    uart UART(
        .CLK(CLK), .RST(RST),
        .CE(CEUART), .WE(WE), .SEL(SEL),
        .DATI(DATO), .DATO(DATUART),
        .RXD(RXD), .TXD(TXD), .CTS(CTS), .RTS(RTS)
    );

    sys SYS(
        .CLK_SRC(CLK_SRC), .CLK(CLK), .SLP(SLP), .WAKEUP(~KEYXI[4]), .RST(RST),
        .CE(CESYS), .WE(WE), .SEL(SEL), .ACK(ACK),
        .DATI(DATO), .DATO(DATSYS),
        .EVENT_REQ(EVENT_REQ),
        .EVENT_ACK(EVENT_ACK),
```

```
    .EVENT_INFO(EVENT_INFO),
    .STB(STB), .ADR(ADR)
  );

toptotal COP(
    .clk(CLK),  .rst(RST), .address(ADR[15:0]),
    .dati(DATO), .cesum(CESUM), .ceres(CERES),
    .cethr(CETHR), .ceconv(CECONV), .cs_0(CEDPRAM),
    .we_0(WE),  .sel(SEL),  .dato(DATCOP)
  );
```

...

Cabe destacar del código anterior la correspondencia, como es lógico, de los distintos puertos de entrada y salida del bloque *toptotal* que hemos incluido en el fichero *top.v*, con los de la figura 3.19, que mostraba el coprocesador como una caja negra.

Otro aspecto a tratar en este punto es la correcta conexión entre los puertos de entrada y salida del procesador con los de la memoria de doble puerto, contenida en el coprocesador. Es muy importante esta unión debido a que son estas señales las que van a unirse al integrar el coprocesador a la estructura de Aquarius. El problema es que los puertos de la memoria de doble puerto son de 8bit de ancho y los de Aquarius lo son de 32bit. Para solucionar el aparente problema lo que debemos hacer es estudiar el comportamiento de una señal que resulta clave en este tipo de conexiones. Hablamos de la señal SEL. Esta señal, de 4bit de ancho, nos indica la posición de datos válida, es decir, cuáles de los 4 bytes en los que está dividido un puerto de 32bit van a tener información útil.

Para ver cómo funciona en el caso que nos interesa, es decir, en la unión de Aquarius con un dispositivo de 8bit, vamos a ver cómo se comporta la señal SEL al realizar varios accesos consecutivos en lectura a un dispositivo de estas características (memoria de doble puerto):

ADR [31:0]	**SEL [3:0]**	**DATCOP_out [31:0]**	**DATI [31:0]**
0xFFFD000A	0010	0x0000**00**00	0x00000000
0xFFFD000B	0001	0x000000**FF**	0x000000FF
0xFFFD000C	1000	0x**00**000000	0x00000000
0xFFFD000D	0100	0x00**00**00FF	0x00000000
0xFFFD000E	0010	0x0000**00**00	0x00000000
0xFFFD000F	0001	0x00000**00**	0x00000000
0xFFFD0010	1000	0x**00**000000	0x00000000
0xFFFD0011	0100	0x00**00**00FF	0x00000000
0xFFFD0012	0010	0x0000**00**FF	0x00000000
0xFFFD0013	...	...	

Tabla 3.13. Ejemplo de acceso de Aquarius a dispositivo de 8bit de arquitectura interna

La tabla anterior nos muestra el resultado de unir directamente el bus de datos de entrada a Aquarius, a la salida de la memoria de doble puerto a través de la señal DATCOP_out. Se realiza sencillamente conectando los 8bit de datos de salida de la memoria de doble puerto a los 8bits menos significativos de la señal DATCOP, que es la que conectamos a DATI, de Aquarius. El problema es que no opera como nosotros deseamos.

La tabla muestra el contenido de las señales que conectamos, así como el de la señal SEL. La columna correspondiente a DATCOP_out muestra los datos de salida de la memoria de doble puerto, que están colocados en el byte menos significativo de la señal en cuestión. A su derecha, vemos qué es lo que realmente lee Aquarius. Lo que ocurre es que lo que lee Aquarius es lo que le indica la señal SEL, es decir, lee los bytes señalados en negrita en la tabla. Por tanto, no se debe conectar directamente la salida de la memoria de doble puerto a los 8bits menos significativos de DATCOP_out (en el fichero *toptotal*.v, esta señal se llama *dato*, pero usamos este nombre para no confundir con las de Aquarius). La forma correcta de realizar esta conexión es la siguiente, tal como se incluye en el fichero *toptotal.v*:

...

```
always @(cs_0 or datmem or sel or we_0)
   begin
      if ((cs_0 == 1'b1) && (we_0==1'b0) && (sel[3:0]==4'b1000))
         begin
            dato[31:24] <= datmem;
            dato[23:16] <= 8'h00;
            dato[15:8] <= 8'h00;
             dato[7:0] <= 8'h00;
         end
     else if ((cs_0 == 1'b1) && (we_0==1'b0) && (sel[3:0]==4'b0100))
         begin
            dato[31:24] <= 8'h00;
            dato[23:16] <= datmem;
             dato[15:8] <= 8'h00;
             dato[7:0] <= 8'h00;
         end
     else if ((cs_0 == 1'b1) && (we_0==1'b0) && (sel[3:0]==4'b0010))
         begin
            dato[31:24] <= 8'h00;
            dato[23:16] <= 8'h00;
             dato[15:8] <= datmem;
             dato[7:0] <= 8'h00;
          end
      else if ((cs_0 == 1'b1) && (we_0==1'b0) && (sel[3:0]==4'b0001))
         begin
            dato[31:24] <= 8'h00;
            dato[23:16] <= 8'h00;
            dato[15:8] <= 8'h00;
             dato[7:0] <= datmem;
          end
       else
         begin
            dato[31:0] <= 32'h00000000;
         end
   end
```

...

Lo que hay que entender del código anterior es que vamos a colocar el dato de salida en la posición que indique SEL, garantizando así que Aquarius va a leer el dato deseado correctamente.

La tabla quedaría:

ADR [31:0]	SEL [3:0]	DATCOP_out [31:0]	DATI [31:0]
0xFFFD000A	0010	0x0000**00**00	0x00000000
0xFFFD000B	0001	0x000000**FF**	0x000000FF
0xFFFD000C	1000	0x**00**000000	0x00000000
0xFFFD000D	0100	0x00**FF**0000	0x00FF0000
0xFFFD000E	0010	0x0000**00**00	0x00000000
0xFFFD000F	0001	0x00000**00**	0x00000000
0xFFFD0010	1000	0x**00**000000	0x00000000
0xFFFD0011	0100	0x00**FF**0000	0x00FF0000
0xFFFD0012	0010	0x0000**FF**00	0x0000FF00
0xFFFD0013	...	...	

Tabla 3.14. Ejemplo corregido de acceso a dispositivo con arquitectura interna de 8bits.

Para los accesos en escritura de Aquarius a la memoria de doble puerto no existe problema alguno, ya que lo que realiza el procesador es replicar el dato útil a lo largo del bus de datos, asegurando que, sea cual sea el valor de la señal SEL, el dato escrito es el correcto. Lo vemos con un ejemplo:

ADR [31:0]	SEL [3:0]	DATO [31:0]	DATCOP_in [31:0]
0xFFFD000A	0010	0x00000**00**	0x00000000
0xFFFD000B	0001	0xFFFFFF**FF**	0x000000FF
0xFFFD000C	1000	0x00000**00**	0x00000000
0xFFFD000D	0100	0xFFFFFF**FF**	0x000000FF
0xFFFD000E	0010	0x00000**00**	0x00000000
0xFFFD000F	0001	0x00000**00**	0x00000000
0xFFFD0010	1000	0x00000**00**	0x00000000
0xFFFD0011	0100	0xFFFFFF**FF**	0x000000FF
0xFFFD0012	0010	0xFFFFFF**FF**	0x000000FF
0xFFFD0013	...	...	

Tabla 3.15. Acceso en escritura de Aquarius al coprocesador.

Lo realmente interesante de este tipo de acceso por parte de Aquarius es que, al replicar el dato útil a lo ancho del bus de datos DATO, nos basta con seleccionar los menos significativos (en negrita en la tabla), independientemente de la señal SEL, lo que simplifica la lógica de selección de la posición de datos válida y, por tanto, reduce la circuitería necesaria para un correcto funcionamiento.

Adaptación del software

Pero aún no está todo dicho con respecto a la integración del coprocesador visual en el sistema procesador. Lo está desde el punto de vista **hardware**, pero no desde el punto de vista **software**.

Como vimos en el apartado *3.2.2*, referido a la programación de aplicaciones en Aquarius, el usuario programaba la aplicación deseada en la ROM interna del

procesador a partir de un código escrito en lenguaje C. Ese código estaba acompañado de un fichero de cabecera, llamado "*common.h*", en el que eran definidos, entre otras cosas, cómo eran los periféricos internamente, qué registros tenían y qué dirección base tenían dentro del mapa de memoria del sistema. Veamos un extracto del fichero "*common.h*" correspondiente a Aquarius original:

```
                    ...

struct st_sys                       /*Definición del controlador del sistema*/
{
   unsigned long  INTCTL;           /*Registros internos*/
   unsigned short BRKADR;
};


#define PORTO (*(volatile struct st_porto  )0xabcd0000)     /*Definición de   */
#define PORTI (*(volatile struct st_porti  )0xabcd0000)     /*direcciones base*/
#define UART  (*(volatile struct st_uart   *)0xabcd0100)
#define SYS   (*(volatile struct st_sys    *)0xabcd0200)

                    ...
```

En el fragmento de texto anterior puede verse cómo se define un periférico de Aquarius, el controlador del sistema, mediante una estructura que contiene sus dos registros internos. En este caso es bastante simple, ya que es el periférico más sencillo desde el punto de vista estructural. Por otra parte, vemos la definición de las direcciones base de los distintos periféricos originales de Aquarius.

Vemos ahora cómo queda el mismo fragmento de código tras incluir el sistema coprocesador con cada uno de sus subsistemas:

```
                    ...

struct st_sys               /*Definición del controlador del stma como estructura*/
{
   unsigned long  INTCTL;       /*Registros internos del controlador del stma*/
   unsigned short BRKADR;
};

struct st_conv                      /*Definición del convolucionador como estructura*/
{
   signed char MASK0;               /*Registros internos del convolucionador*/
   signed char MASK1;
   signed char MASK2;
   signed char MASK3;
   signed char MASK4;
   signed char MASK5;
   signed char MASK6;
   signed char MASK7;
   signed char MASK8;
   unsigned char INICIOCONV;
   unsigned short DO;
   unsigned char M;
   unsigned char N;
   unsigned short DD;


};
```

```
struct st_sum                    /*Definición del sumador como estructura*/
{
   unsigned char M;              /*Registros internos del sumador*/
   unsigned char N;
   unsigned short DO1;
   unsigned short DO2;
   unsigned short DD;
   unsigned char INICIOSUM;

};

struct st_thr                    /*Definición del umbralizador como estructura*/
{
   unsigned char M;              /*Registros internos del umbralizador*/
   unsigned char N;
   unsigned short DO;
   unsigned short DD;
   unsigned char INICIOTHR;
   unsigned char T;

};


struct st_res                    /*Definición del restador como estructura*/
{
   unsigned char M;              /*Registros internos del restador*/
   unsigned char N;
   unsigned short DO1;
   unsigned short DO2;
   unsigned short DD;
   unsigned char INICIORES;

};


struct st_dpram                  /*Definición de la memoria de doble puerto como */
{                                /*estructura                                    */
   unsigned char MEMO[65536];    /*Registros que forman la memoria*/
};

#define PORTO (*(volatile struct st_porto  )0xabcd0000)      /*Conjunto de direcciones*/
#define PORTI (*(volatile struct st_porti  )0xabcd0000)      /*base del sistema       */

#define UART  (*(volatile struct st_uart   *)0xabcd0100)
#define SYS   (*(volatile struct st_sys    *)0xabcd0200)
#define CONV (*(volatile struct st_conv  *)0xabcd0300)
#define SUM  (*(volatile struct st_sum   *)0xabcd0400)
#define THR   (*(volatile struct st_thr    *)0xabcd0500)
#define RES   (*(volatile struct st_res    *)0xabcd0600)
#define DPRAM   (*(volatile struct st_dpram    *)0xfffd0000)
                  ...
```

En este extracto de código, vemos cómo cada subsistema ha sido incluido del mismo modo que se incluyeron los periféricos originales de Aquarius, contemplando todos los registros internos de los que se componen y definiendo la dirección base de cada uno de ellos de acuerdo con sus respectivas disposiciones dentro del mapa de memoria modificado del sistema procesador, tal como aparecía reflejado en la tabla 3.12.

A continuación explicaremos en detalle la estructura y funcionamiento de cada uno de los subsistemas que componen el coprocesador, y que aparecían en el esquema interno (figura 3.17), una vez que hemos visto cómo se ha conectado a Aquarius.

3.3.3 Convolucionador

Esquema del convolucionador

La descripción de la estructura interna así como de los puertos de entrada / salida del convolucionador se muestra en la siguiente figura:

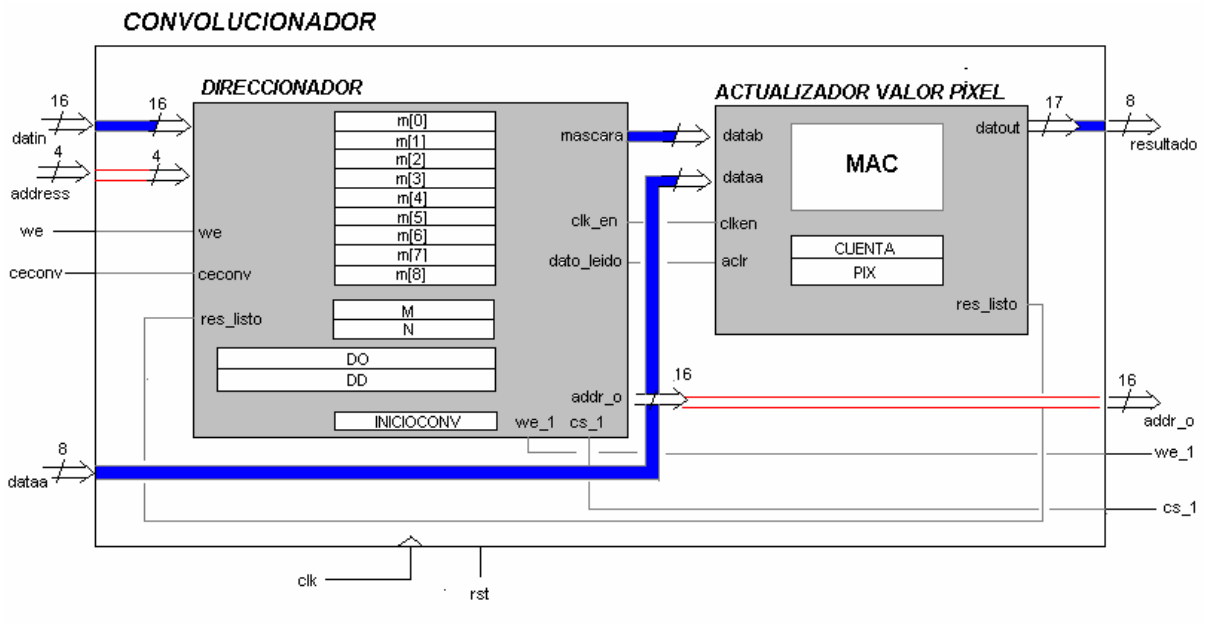


Figura 3.20. Estructura del convolucionador.

La figura anterior muestra la estructura interna del convolucionador, mostrando además su interfaz de entrada y salida. Pasemos a describir los puertos del subsistema:

- **DATIN**: Entrada de datos procedente de Aquarius. En este caso servirá para configurar los registros internos del subsistema, que serán descritos más adelante. Su ancho es de 16bit.
- **ADDRESS**: Entrada de direcciones desde Aquarius. Su ancho es de 4bit y se corresponde con los 4 bits menos significativos del bus de direcciones del procesador.
- **WE**: Señal que indica si el ciclo actual es de escritura (cuando vale 1) o de lectura (si su valor es 0). Procede de Aquarius.
- **CECONV**: Señal de habilitación del convolucionador. Procede de Aquarius y se activa cuando el procesador accede a los registros internos del subsistema.
- **DATAA**: Bus de datos de 8 bits procedentes de la memoria de doble puerto. Es una señal de entrada.
- **CLK**: Reloj del sistema.
- **RST**: Señal de reset.
- **ADDR_O**: Bus de direcciones de la memoria de doble puerto. Es una señal de salida.
- **WE_1**: Señal que indica si re realiza una operación de escritura (1) o lectura (0) al acceder a la memoria de doble puerto.

- **CS_1**: Señal que habilita la memoria de doble puerto cuando el convolucionador necesita acceder a ella.
- **RESULTADO**: Valor de 8 bits devuelto correspondiente a la respuesta de la máscara al hacerla pasar por un píxel. Este valor se almacena en la DPRAM. Es una señal de salida.

Estructura interna del convolucionador

El convolucionador, como puede verse en la figura 3.20, consta de dos bloques:

- **Direccionador:** encargado de escanear la imagen original, recavando los valores de los píxeles vecinos necesarios para realizar la convolución.
- **Actualizador del valor del píxel (AVP):** se encarga de obtener el resultado de procesar un píxel de la imagen original. Es una especie de MAC (*multiply-accumulate*), adaptado para pasar toda la máscara de 3x3. Este MAC opera con números positivos y negativos para resultados parciales, aunque presenta a la salida un valor sin signo representable por 8 bits, adecuado para ser guardado en la DPRAM.

Vemos ahora la interfaz de E/S del *direccionador*:

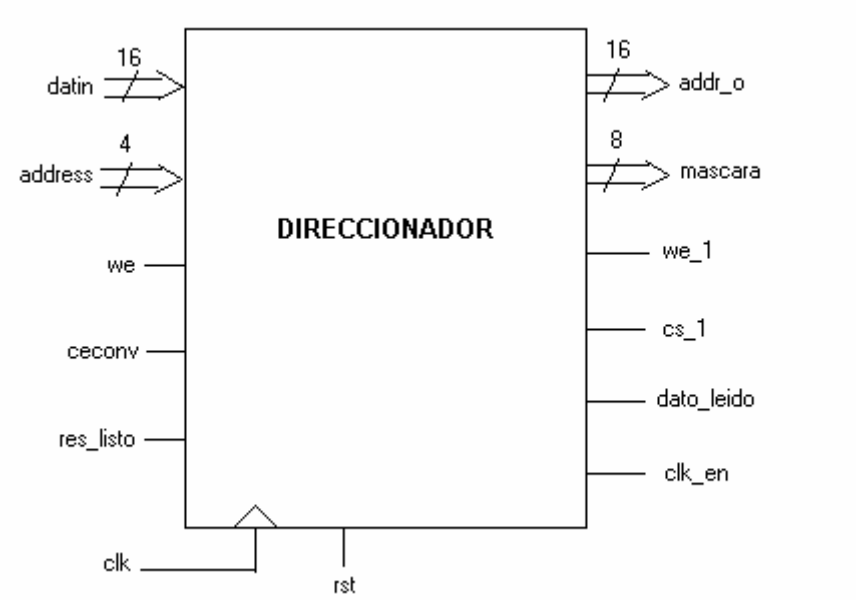


Figura 3.21. Interfaz de E/S del direccionador.

Describimos los puertos que no integran el conjunto de puertos de E/S del convolucionador:

- **RES_LISTO:** Señal procedente del *actualizador del valor del píxel*. Indica que ha terminado de realizar una convolución sobre un píxel. Es una señal de entrada.
- **MASCARA:** Datos de la máscara usada por el *direccionador* y que éste pasa al *AVP* para realizar la operación.
- **DATO_LEIDO:** Señal que se envía al *AVP* para resetearlo de manera síncrona.

- **CLK_EN:** Señal que habilita o deshabilita el funcionamiento del *AVP*.

A continuación vamos a mostrar los registros internos del direccionador, que son los que permiten el correcto funcionamiento del convolucionador, ya que Aquarius escribe en ellos para configurar su funcionamiento:

[CONV] Address=0xABCD0300 R/W MASK[0]

M07	M06	M05	M04	M03	M02	M01	M00

[CONV] Address=0xABCD0301 R/W MASK[1]

M17	M16	M15	M14	M13	M12	M11	M10

[CONV] Address=0xABCD0302 R/W MASK[2]

M27	M26	M25	M24	M23	M22	M21	M20

[CONV] Address=0xABCD0303 R/W MASK[3]

M37	M36	M35	M34	M33	M32	M31	M30

[CONV] Address=0xABCD0304 R/W MASK[4]

M47	M46	M45	M44	M43	M42	M41	M40

[CONV] Address=0xABCD0305 R/W MASK[5]

M57	M56	M55	M54	M53	M52	M51	M50

[CONV] Address=0xABCD0306 R/W MASK[6]

M67	M66	M65	M64	M63	M62	M61	M60

[CONV] Address=0xABCD0307 R/W MASK[7]

M77	M76	M75	M74	M73	M72	M71	M70

[CONV] Address=0xABCD0308 R/W MASK[8]

M87	M86	M85	M84	M83	M82	M81	M80

[CONV] Address=0xABCD0309 R/W INICIOCONV

Res.	Res.	Res.	Res.	Res.	Res.	Res.	Inic.

[CONV] Address=0xABCD0310 R/W DO (Direcc. Origen)

DO15	DO14	DO13	DO12	DO11	DO10	DO9	DO8
D07	DO6	DO5	DO4	DO3	DO2	DO1	DO0

[CONV] Address=0xABCD0312 R/W M

M7	M6	M5	M4	M3	M2	M1	M0

[CONV] Address=0xABCD0313 R/W N

N7	N6	N5	N4	N3	N2	N1	N0

[CONV] Address=0xABCD0314 R/W DD (Dir. Destino)

DD15	DD14	DD13	DD12	DD11	DD10	DD9	DD8
DD7	DD6	DD5	DD4	DD3	DD2	DD1	DD0

Tabla 3.16. Registros internos del direcccionador.

Pasamos a ver en detalle los registros:

- **MASK [8:0]**: Conjunto de 9 registros de 8 bits cada uno que contienen los 9 elementos de la máscara 3x3. Los valores están comprendidos entre [-128,127], al ser de 8 bits con signo.
- **INICIOCONV**: Bit menos significativo que pone a uno lógico Aquarius para que el convolucionador comience a operar. El resto de bits del registro (bits del 1 al 7) están reservados (res.).
- **DO**: Dirección de 16 bits de la memoria de doble puerto que indica el comienzo de la imagen a procesar. Es un valor que introduce Aquarius.

- **M**: Una de las dimensiones de la imagen, correspondiente al número de filas. También es Aquarius quien escribe su valor.
- **N**: El número de columnas de la imagen. Aquarius la escribe.
- **DD**: Dirección de 16 bits de la memoria de doble puerto en donde se va a escribir la imagen procesada.

El *actualizador del valor del píxel (AVP)*, posee la siguiente interfaz de E/S:

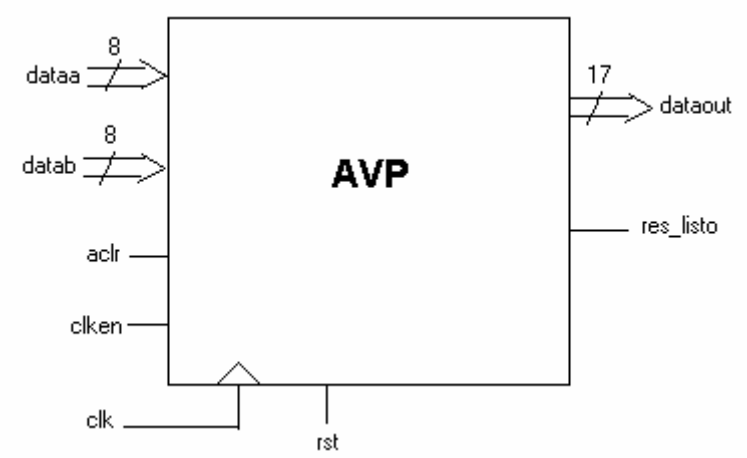


Figura 3.22. Interfaz de E/S del actualizador del valor del píxel.

Describimos los puertos que no formaban parte de la interfaz de E/S del convolucionador:

- **DATAB**: Línea de datos de entrada al subsistema. Su tamaño es de 8 bits con signo. Los datos entrantes corresponden a los coeficientes de la máscara de procesado.
- **ACLR**: Señal que pone a cero el acumulador del MAC interno del *AVP*. Equivale a un reset síncrono.
- **CLKEN**: Señal de habilitación del reloj. Es la que hace que el MAC interno funcione o se quede suspendido.
- **RES_LISTO**: Señal que indica que el *AVP* posee en su salida el resultado de realizar 9 operaciones (lo correspondiente a la determinar la repuesta de una máscara).

Respecto a la estructura interna de *AVP*, hay que destacar que va a estar condicionada de manera estrecha a la aplicación a desarrollar por parte del sistema de visión. Esto se debe a que la salida de datos del *AVP* será diferente según se programe, por ejemplo, una operación de convolución persiguiendo el cálculo de una imagen gradiente, o una operación de convolución para el cálculo de una difusión en una operación de simulación de una CNN, como vimos en el apartado 2.2.2. Como veremos más adelante, el ejemplo de operación que llevaremos a cabo será una simulación de una CNN, por lo que la estructura del hardware del *AVP* estará adaptada para ese propósito. Esto constituye una de las limitaciones del sistema.

De este modo, a la salida de datos del *AVP*, tendremos un valor de píxel procesado dependiente del antiguo valor del píxel sin procesar, como indicaba la ecuación (2.50), y la aportación de la respuesta de la máscara de difusión,

ecuación (2.43), ponderada por la razón entre el paso de integración, Δt, y la constante de tiempo de la red difusiva τ. El valor de esta razón, que pondera la respuesta de la máscara, debe ser fijado, por hardware, a un número potencia de 2, implementándose esta división mediante un registro de desplazamiento, de manera que no se complique el hardware. Esto es otra limitación del sistema.

El *AVP* realiza el procesado por máscaras de imágenes cuyos píxeles poseen una gama de valores posibles de niveles de gris correspondientes a 8 bits de ancho, es decir 256 niveles de gris posibles. Por tanto, en este caso concreto, hay que destacar las siguientes características:

- Una línea de datos, DATAA, va a albergar valores sin signo procedentes de una memoria de ancho de 8 bits, correspondientes a los valores del nivel de gris de los píxeles de la imagen a tratar.
- La otra línea de datos, DATAB, va a presentar valores con signo de tamaño 8 bits correspondientes a los valores de los elementos que componen la máscara de procesado.
- **CUENTA:** Registro interno que ayuda al *AVP* a controlar el número de operaciones realizadas por el MAC interno.
- **PIX:** Registro interno que almacena el valor del píxel cuyo valor se está actualizando.
- El número de ciclos del procesado viene dado por el tiempo que se tarda en realizar esas 9 multiplicaciones con sus consiguientes acumulaciones.
- El resultado final debe ser un valor que se encuentre en el intervalo [0,255], debido a que debe ser almacenado en una memoria de 8 bits de ancho.

Un detalle que complica la estructura del sistema es que los resultados parciales no se pueden ni manipular, ni acotar, independientemente del tamaño que deba tener el resultado final, ya que ello afectaría seriamente al resultado obtenido. Es por ello que el *AVP* va a presentar internamente unos registros de tamaños no correspondidos en el exterior.

Lo primero a tener en cuenta es que el dato procedente de la memoria de doble puerto es sin signo, pero debe ser tratado como un valor con signo para que el MAC interno opere correctamente. Para ello, este valor se almacena internamente en un registro de 9 bits, en el que el bit más significativo vale 0, asegurando que, sea cual sea el valor que introducimos en él, tenga signo positivo. Como el segundo operando del *AVP* es de 8 bits con signo, tendríamos que, tanto los resultados parciales como el final, deben almacenarse en registros de 17 bits. Cuando el *AVP* activa la señal RES_LISTO, y sólo entonces, es cuando truncamos el contenido del acumulador a un valor de 8 bits **sin signo**, que es un valor correcto para ser almacenado en memoria como nivel de gris de un píxel, para lo cual hay que tomar valores absolutos y posteriormente truncar a 8 bits.

Después de realizar el procesado de un píxel, es el direccionador el que fuerza un reset síncrono en el *AVP*, para limpiar el contenido del acumulador del MAC interno y dejarlo listo para una próxima operación.

Funcionamiento del convolucionador

El procesamiento que el convolucionador realiza sobre la imagen tiene lugar en el dominio espacial, actuando directamente sobre los píxeles, mediante el paso de una máscara espacial de 3x3 elementos (programable por software) que genera un resultado dependiente tanto del valor del píxel a procesar como de sus vecinos.

El bloque *direccionador* debe entregarle los datos precisos al *actualizador del valor del píxel*, es decir, direcciona la posición adecuada de la memoria de doble puerto a través de la señal ADDR_O, haciendo que la memoria devuelva valores del nivel de gris de píxeles de la imagen a procesar y, por otro lado, y a través de la señal MASCARA, pasarle el valor del elemento de la máscara correspondiente al píxel de la imagen.

El *actualizador del valor del píxel* realiza una operación de multiplicación y acumulación sobre los valores que recibe de la DPRAM y del *direccionador*, y modifica el valor del píxel procesado según el valor obtenido de la operación del MAC interno que posee. Esto lo realiza de forma ponderada, como se indica en la ecuación (2.50).

Como hemos dicho durante la descripción de los diferentes registros que conforman la estructura interna del convolucionador, lo primero que debe ocurrir para que comience la operación del subsistema es que Aquarius configure todos y cada uno de los registros mostrados en la tabla 3.16. Una vez escritos todos los registros con información útil, el siguiente paso es que Aquarius escriba un 1 en el bit menos significativo del registro INICIOCONV. Esa acción lanza el comienzo de la actividad del subsistema.

Vemos, en el diagrama de estados mostrado a continuación, el funcionamiento del sistema una vez configurado:

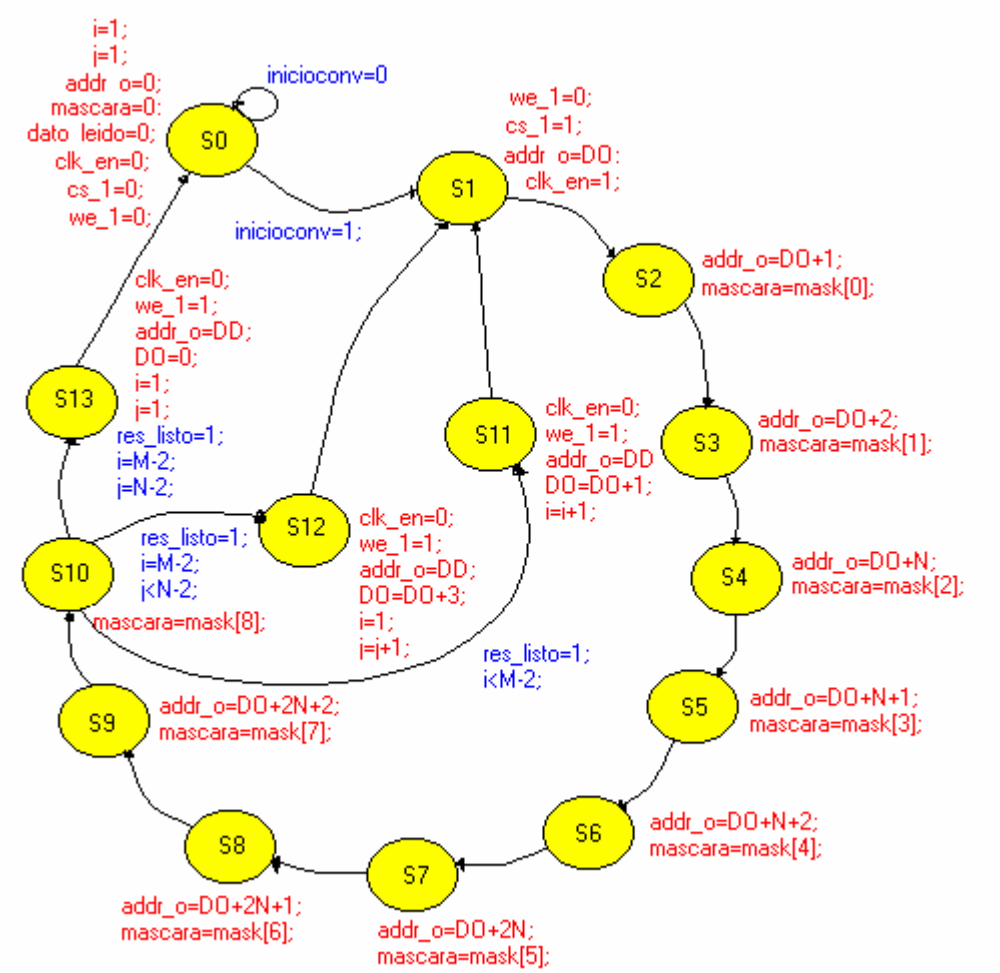


Figura 3.23. Diagrama de estados del convolucionador.

Vamos a ver a continuación cómo se realiza el paso de la máscara por un píxel en el diagrama de estados de la figura anterior. Para ello el *direccionador* debe pasarle los datos precisos al *AVP*, es decir, direccionar la posición adecuada de la memoria de doble puerto a través de la señal ADDR_O, haciendo que la memoria devuelva valores del nivel de gris de píxeles de la imagen a procesar y, por otro lado, y a través de la señal MASCARA, pasarle el valor del elemento de la máscara correspondiente al píxel de la imagen.

- ***S0***: El primer estado se corresponde con el estado de reposo. En él es importante que las salidas que van a parar a una puerta OR de entrada a la memoria de doble puerto, es decir, ADDR_O, CS_1 y WE_1 (ver esquema interno del coprocesador, figura 3.17), tengan un valor igual a cero, debido a que de esta forma se permite el acceso de cualquier otro subsistema a la memoria. Permaneceremos en S0 hasta que Aquarius escriba un 1 en INICIOCONV.

- ***S1***: En este estado se activa la línea de habilitación de la memoria de doble puerto, tomando el *direccionador* control sobre la misma. Se direcciona el primer dato a pasar al *AVP*. Aquí se pone de manifiesto un retraso de un estado entre el dato entregado al *AVP* a través de la memoria y el que se le pasa directamente a través de la línea MASCARA, que aún no se produce en S1,

debido al tiempo que tarda la memoria en sacar el dato direccionado a su salida de manera síncrona.

- ***S2, S3, S4, S5, S6, S7, S8, S9***: Este proceso de entrega de datos al *AVP* se realiza nueve veces (tantos como elementos contiene una máscara 3x3). Durante todo este procedimiento permanece activa la señal CLK_EN, lo que permite al MAC interno del *AVP* continuar operando.

- ***S10***: Estado en el que no se direcciona ningún dato de nuevo de memoria de doble puerto, pero en el que, debido al desfase anteriormente comentado, se pasa el último elemento de la máscara al *AVP*.

- ***S11,S12***: En estos estados el MAC interno del *AVP* ya ha concluido su operación por lo que CLK_EN se anula, se escribe el dato resultante de la multiplicación / acumulado en la dirección señalada por ADDR_O, y se actualizan los cursores i y j, que llevan el control de la posición del píxel a procesar dentro de la imagen. Son dos estados diferentes porque es necesario distinguir cuándo se está actuando sobre un píxel que cae al final de una fila de la imagen (S12) y cuándo se actúa sobre uno estándar (S11). El MAC interno del *AVP* es reseteado en ambos estados, preparándose para realizar otra operación. Ambos estados desembocan en S1, comenzando el proceso de un nuevo píxel, indicado por la actualización de DO.

- ***S12***: Estado final del procesado de una imagen. Es parecido a los estados S11 y S12, con la diferencia de que el estado siguiente es S0, quedando el dispositivo en reposo.

Una consideración necesaria acerca del retraso de un ciclo anteriormente detallado es que asumimos una aparente falta de uniformidad a la hora de definir los estados en los que existe un paso de datos al *AVP* (en un estado se ofrece al *AVP* el dato correspondiente al elemento *i* de la máscara mientras que se direcciona el píxel correspondiente a la posición del elemento *i+1* de la máscara) a cambio de reducir en un ciclo el tiempo de procesado por píxel. Este hecho resulta importante cuando hablamos de aplicaciones de procesado en tiempo real, donde ahorrarse un ciclo por píxel puede llegar a ser clave.

Es importante recalcar, llegados a este punto, que el convolucionador no opera sobre toda la imagen. Deja un margen de 1 píxel sin procesar, para que la máscara permanezca siempre íntegra dentro de la imagen. Si hubiéramos optado por procesar la imagen completa, el coste computacional hubiera sido alto, ya que existen distintos casos en los que la máscara no cae de manera completa dentro de la imagen:

- Esquina superior izquierda,
- Esquina superior derecha,
- Esquina inferior izquierda,
- Esquina inferior derecha,
- Elementos situados en la fila superior, salvo en las esquinas,
- Elementos situados en la fila inferior salvo en las esquinas,
- Elementos situados en la primera columna, salvo en las esquinas,
- Elementos situados en la última columna, salvo en las esquinas.

Todos estos casos expuestos anteriormente dan lugar a distintas configuraciones de procesado que habría que considerar separadamente para que el procesado fuese completo, lo que conlleva un incremento bastante considerable en la complejidad del sistema sin que desde el punto de vista del

procesamiento de la imagen se ganara mucho en cuanto a la información contenida en la imagen resultado. Además, el efecto en imágenes de un tamaño mínimamente aceptable (32x32, por ejemplo) apenas es importante y acarrea, como es lógico, un ahorro de tiempo de procesado, ya que, para una imagen de 32x32 píxeles, estaríamos procesando realmente 30x30 píxeles.

3.3.4 Sumador / Restador

Este bloque es otro de los subsistemas del coprocesador visual que se encarga de sumar o restar dos imágenes, generando otra imagen del mismo tamaño que las originales (suponemos que son del mismo tamaño) y que llamaremos imagen suma o imagen diferencia, según sea el caso.

Este bloque es independiente de los vistos anteriormente y sólo se comunica con la memoria de doble puerto y con Aquarius, y está compuesto a su vez de dos subsistemas, el sumador y el restador. Los vemos por separado:

Sumador de imágenes

La siguiente figura muestra un esquema del sumador de imágenes:

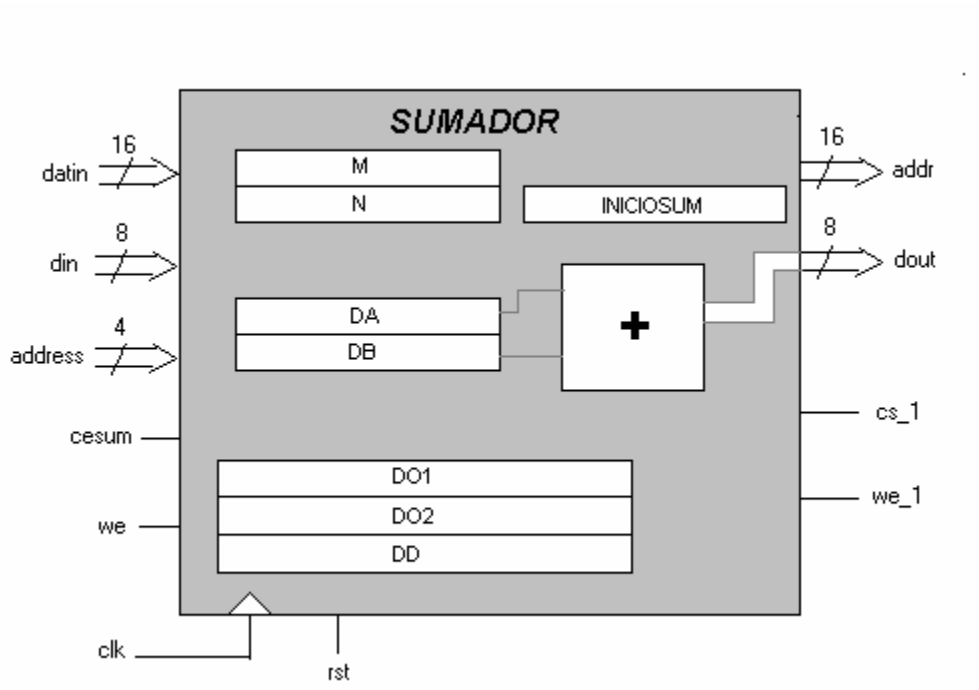


Figura 3.24. Esquema del sumador de imágenes.

Estudiamos sus puertos de entrada / salida:

- **DATIN**: Línea de datos procedente de Aquarius dedicada a la configuración de los registros internos del sumador por parte de aquél.
- **DIN**: Línea de entrada de datos procedentes de la memoria de doble puerto.

- **ADDRESS**: Entrada de direcciones desde Aquarius. Su ancho es de 4 bits y se corresponden con los 4 bits menos significativos del bus de direcciones del procesador.
- **CESUM**: Señal de habilitación del sumador de imágenes. Procede de Aquarius y se activa cundo éste quiere programar los registros internos del subsistema.
- **WE**: Señal que indica si el ciclo de acceso al subsistema en de escritura (1) o de lectura(0). Procede de Aquarius.
- **CLK**: Reloj del sistema.
- **RST**: Reset del sistema.
- **ADDR**: Línea de direcciones de 16 bits de salida. A través de ella, el sumador indica la dirección de la memoria de doble puerto en la que va a escribir o de la que va a leer un dato.
- **DOUT**: Línea de datos de 8 bits de salida. Su destino es la memoria de doble puerto, donde se almacenará la imagen suma.
- **CS_1**: Señal de habilitación de la memoria de doble puerto (DPRAM). Es una señal de salida.
- **WE_1**: Señal que indica si el ciclo de acceso a la DPRAM es de escritura (1) o de lectura (0). Es una señal de salida.

Estructura interna del sumador de imágenes

El sumador de imágenes posee unos registros configurables por parte de Aquarius que se detallan a continuación:

[SUM] Address=0xABCD0400 R/W M

M7	M6	M5	M4	M3	M2	M1	M0

[SUM] Address=0xABCD0401 R/W N

N7	N6	N5	N4	N3	N2	N1	N0

[SUM] Address=0xABCD0402 R/W DO1 (Dir. Origen Imagen 1)

DO115	DO114	DO113	DO112	DO111	DO110	DO19	DO18
DO17	DO16	DO15	DO14	DO13	DO12	DO11	DO10

[SUM] Address=0xABCD0404 R/W DO2 (Direcc. Origen Imagen 2)

DO215	DO214	DO213	DO212	DO211	DO210	DO29	DO28
D027	DO26	DO25	DO24	DO23	DO22	DO21	DO20

[SUM] Address=0xABCD0406 R/W DD (Direcc. Destino)

DD15	DD14	DD13	DD12	DD11	DD10	DD9	DD8
DD7	DD6	DD5	DD4	DD3	DD2	DD1	DD0

[SUM] Address=0xABCD0408 R/W INICIOSUM

Res.	Res.	Res.	Res.	Res.	Res.	Res.	Inic.

Tabla 3.17. Registros internos del sumador de imágenes.

Detallamos cada uno de estos registros:

- **M**: Una de las dimensiones de la imagen. Aquarius es quien escribe su valor.
- **N**: La otra dimensión de la imagen. Aquarius la escribe.

- **DO1**: Dirección de 16 bits de la memoria de doble puerto que indica el comienzo de una de las dos imágenes a sumar. Es un valor que introduce Aquarius.
- **DO2**: Dirección de 16 bits de la memoria de doble puerto que indica el comienzo de la segunda imagen a sumar. Es un valor que introduce Aquarius.
- **DD**: Dirección de 16 bits de la memoria de doble puerto en donde se va a escribir la imagen suma.
- **INICIOSUM**: Señal que activa Aquarius para que el sumador de imágenes comience a operar.

Funcionamiento del sumador de imágenes

Este subsistema consiste en un bloque dedicado a sumar dos imágenes del mismo modo que se sumarían dos matrices, es decir lo que sumo son los píxeles situados en posiciones idénticas de dos imágenes diferentes. Como el resultado va a almacenarse en la DPRAM, debo truncar el resultado obtenido a 8 bits, estando el resultado dentro del intervalo [0,255].

Una aplicación importante de este subsistema es su utilización en el cálculo del gradiente de una imagen, como vimos en el apartado *2.2.1*, ya que es necesario para ello realizar una suma de las imágenes resultantes de calcular la componente horizontal y vertical del gradiente para obtener la imagen gradiente buscada.

Para que el sumador de imágenes se ponga en funcionamiento es necesaria una previa configuración de los registros internos de éste por parte de Aquarius. Una vez ocurra esto, el sistema estará preparado para operar, y será lanzado por el procesador cuando escriba un 1 en el registro INICIOSUM. El comportamiento del sumador puede verse mejor observando su diagrama de estados:

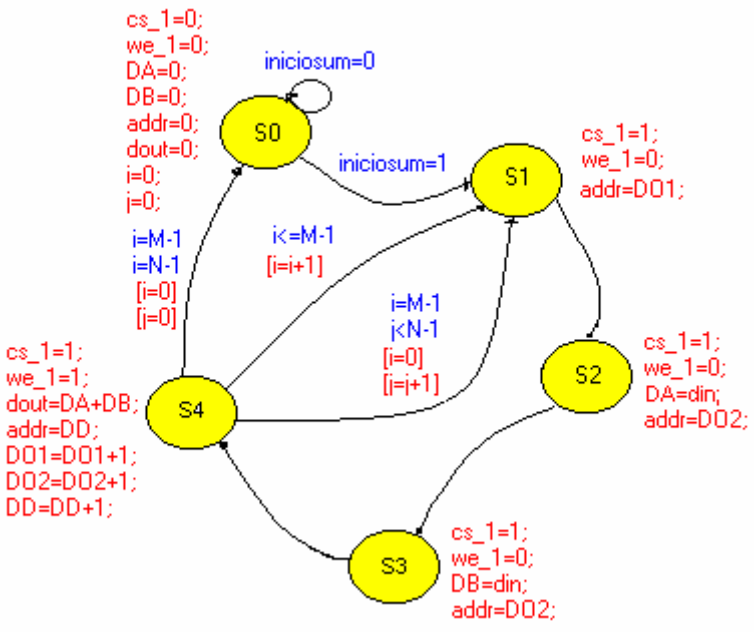


Figura 3.25. Diagrama de estados del sumador de imágenes.

En el diagrama de estados de la figura anterior se detalla la suma de dos píxeles que se encuentran en la misma posición dentro de imágenes distintas dando lugar a otro píxel colocado en la misma posición que aquellos, pero en la imagen suma. Lo vemos estado a estado:

- ***S0***: El primer estado se corresponde con el estado de reposo. En él es importante que las salidas que van a parar a una puerta OR de entrada a la memoria de doble puerto, es decir, ADDR, CS_1 y WE_1 (ver esquema interno del coprocesador, figura 3.17), tengan un valor igual a cero. Permaneceremos en S0 hasta que Aquarius escriba un 1 en INICIOSUM.

- ***S1***: En este estado se activa la línea de habilitación de la memoria de doble puerto, tomando el sumador de imágenes control sobre la misma. Se direcciona el primer sumando con la línea ADDR.

- ***S2***: En este estado, el primer sumando aparece en el puerto DIN, almacenándose en el registro DA, a la vez se cambia ADDR para que direcciones el segundo sumando.

- ***S3***: En este estado, el segundo sumando aparece en DIN y es almacenado en DB.

- ***S4***: En este estado se escribe el dato resultante de la suma de los sumandos (DA+DB) en la dirección señalada por ADDR, teniendo en cuenta que es una suma que satura al valor de 255, se actualizan los cursores i y j, que llevan el control de la posición del píxel a procesar dentro de la imagen y, según el valor de éstos, el ciclo que sigue será S0 o S1. En esta ocasión existen dos transiciones posibles de S4 a S1, ya que la actualización de los cursores dependerá de si nos encontramos en un elemento correspondiente a la última columna de una imagen o no.

En el caso de que nos encontremos en el último píxel de la imagen, el estado siguiente sería S0, volviendo el sistema al reposo inicial.

En esta ocasión la suma se realiza sobre todos los píxeles de las “imágenes sumando”, de ahí que los valores de los cursores vayan desde 0 a M-1 (cursor i) y desde 0 a N-1 (cursor j).

Restador de imágenes

La siguiente figura muestra un esquema del restador de imágenes:

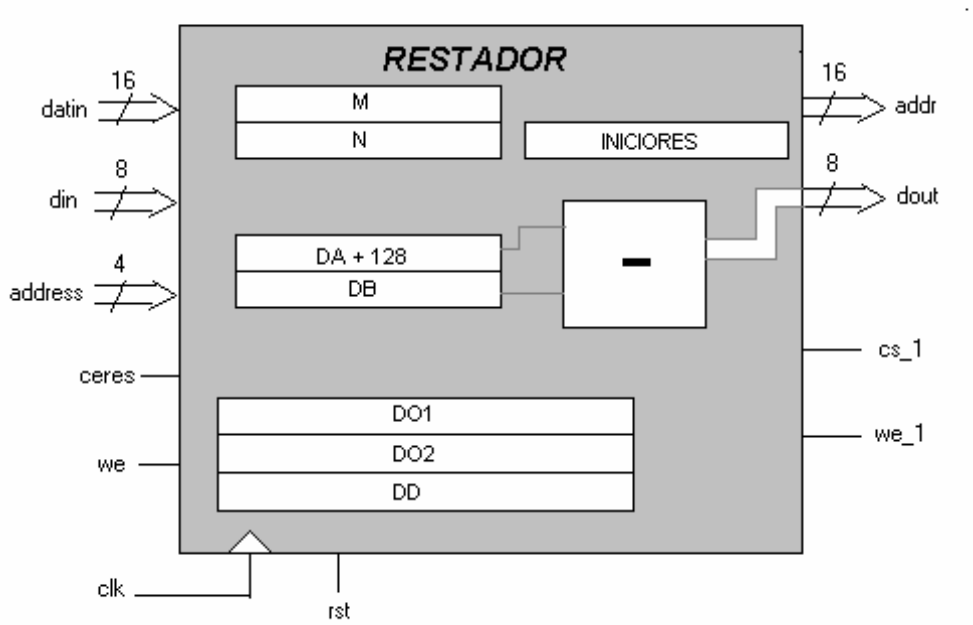


Figura 3.26. Esquema del restador de imágenes.

Estudiamos sus puertos de entrada / salida:

- **DATIN**: Línea de datos procedente de Aquarius dedicada a la configuración de los registros internos del restador por parte de aquél.
- **DIN**: Línea de entrada de datos procedentes de la memoria de doble puerto.
- **ADDRESS**: Entrada de direcciones desde Aquarius. Su ancho es de 4 bits y se corresponden con los 4 bits menos significativos del bus de direcciones del procesador.
- **CERES**: Señal de habilitación del restador de imágenes. Procede de Aquarius y se activa cundo éste quiere programar los registros internos del subsistema.
- **WE**: Señal que indica si el ciclo de acceso al subsistema en de escritura (1) o de lectura(0). Procede de Aquarius.
- **CLK**: Reloj del sistema.
- **RST**: Reset del sistema.
- **ADDR**: Línea de direcciones de 16 bits de salida. A través de ella, el restador indica la dirección de la memoria de doble puerto en la que va a escribir o de la que va a leer un dato.
- **DOUT**: Línea de datos de 8 bits de salida. Su destino es la memoria de doble puerto, donde se almacenará la imagen diferencia.
- **CS_1**: Señal de habilitación de la memoria de doble puerto (DPRAM). Es una señal de salida.

- **WE_1**: Señal que indica si el ciclo de acceso a la DPRAM es de escritura (1) o de lectura (0). Es una señal de salida.

Estructura interna del restador de imágenes

El restador de imágenes posee unos registros configurables por parte de Aquarius que se detallan a continuación:

[RES] Address=0xABCD0600 R/W M

M7	M6	M5	M4	M3	M2	M1	M0

[RES] Address=0xABCD0601 R/W N

N7	N6	N5	N4	N3	N2	N1	N0

[RES] Address=0xABCD0602 R/W DO1 (Dir. Origen Imagen 1)

DO115	DO114	DO113	DO112	DO111	DO110	DO19	DO18
DO17	DO16	DO15	DO14	DO13	DO12	DO11	DO10

[RES] Address=0xABCD0604 R/W DO2 (Direcc. Origen Imagen 2)

DO215	DO214	DO213	DO212	DO211	DO210	DO29	DO28
D027	DO26	DO25	DO24	DO23	DO22	DO21	DO20

[RES] Address=0xABCD0606 R/W DD (Direcc. Destino)

DD15	DD14	DD13	DD12	DD11	DD10	DD9	DD8
DD7	DD6	DD5	DD4	DD3	DD2	DD1	DD0

[RES] Address=0xABCD0608 R/W INICIORES

Res.	Res.	Res.	Res.	Res.	Res.	Res.	Inic.

Tabla 3.18. Registros internos del restador de imágenes.

Detallamos cada uno de estos registros:

- **M**: Una de las dimensiones de la imagen. Aquarius es quien escribe su valor.
- **N**: La otra dimensión de la imagen. Aquarius la escribe.
- **DO1**: Dirección de 16 bits de la memoria de doble puerto que indica el comienzo de una de las dos imágenes a restar. Es un valor que introduce Aquarius.
- **DO2**: Dirección de 16 bits de la memoria de doble puerto que indica el comienzo de la segunda imagen a restar. Es un valor que introduce Aquarius.
- **DD**: Dirección de 16 bits de la memoria de doble puerto en donde se va a escribir la imagen diferencia.
- **INICIORES**: Señal que activa Aquarius para que el restador de imágenes comience a operar.

Funcionamiento del restador de imágenes

Este subsistema consiste en un bloque dedicado a restar dos imágenes del mismo modo que se restarían dos matrices, es decir lo que resto son los píxeles situados en posiciones idénticas de dos imágenes diferentes. Como el resultado va a almacenarse en la DPRAM, debo truncar el resultado obtenido a 8 bits, estando el resultado dentro del intervalo [0,255].

Una aplicación importante de este subsistema es su utilización en la detección de bordes mediante un algoritmo basado en CNN's, como veremos en el

Capítulo 5, ya que va a ser necesario para ello realizar el cálculo de la diferencia de las imágenes resultantes de la difusión con dos tiempos diferentes.

Este bloque posee la peculiaridad de calcular la diferencia entre dos píxeles correspondientes en imágenes diferentes, y después de saturarla superior en inferiormente, la almacena como un número entre 0 y 255. Como nos interesa, por motivos de adaptación de los algoritmos basados en CNN's que manejamos, que las imágenes estén expresadas en un rango de [-1,1], o sea, que el 0 corresponda a lo que hasta ahora ha venido siendo el nivel de gris 128, antes de operar realizaremos una translación del origen y un escalado. El camino de vuelta al rango [0,255] va a necesitar un nuevo escalado que se va a cancelar con el anterior simplificando la implementación. Por tanto vamos a considerar que los píxeles de entrada al sistema fuesen interpretados como números **con signo** de valores comprendidos entre –128 y 127. Para realizar correctamente esta operación es necesario realizar una transformación de variables. Tendíamos que una variable, llamémosla *x*, tiene valores entre 0 y 255, y que otra variable, *y* tiene un rango de valores entre –1 y 1. Las ecuaciones que relacionan estas variables son:

$$y = 1 - x/128 \ ; \tag{3.2}$$

$$x = 128 \cdot (1 - y) \ ; \tag{3.3}$$

De este modo, si *y* alcanza su valor máximo, es decir, 1, *x* tomará aproximadamente su valor mínimo, 0. Si realizo la resta de dos valores en la escala dada por la variable *y*, tendríamos el resultado:

$$y_1 - y_2 = \frac{1}{128}(x_2 - x_1) \ ; \tag{3.4}$$

Lo expresamos en la escala de la variable *x*, que es la que estamos usando en el sistema de visión:

$$x_{resta} = 128 + x_1 - x_2 ; \tag{3.5}$$

Esta expresión es la que usaremos para calcular la imagen diferencia. En ella se observa que el resultado final puede ser superior a 255, saturando a ese valor, e incluso inferior a 0, en cuyo caso saturaría a ese valor. De este modo el resultado final estaría comprendido entre 0 y 255, representable por 8 bits sin signo.

Para que el restador de imágenes se ponga en funcionamiento es necesaria una previa configuración de los registros internos de éste por parte de Aquarius. Una vez ocurra esto, el sistema estará preparado para operar, y será lanzado por el procesador cuando escriba un 1 en el registro INICIORES. El comportamiento del restador puede verse mejor observando su diagrama de estados:

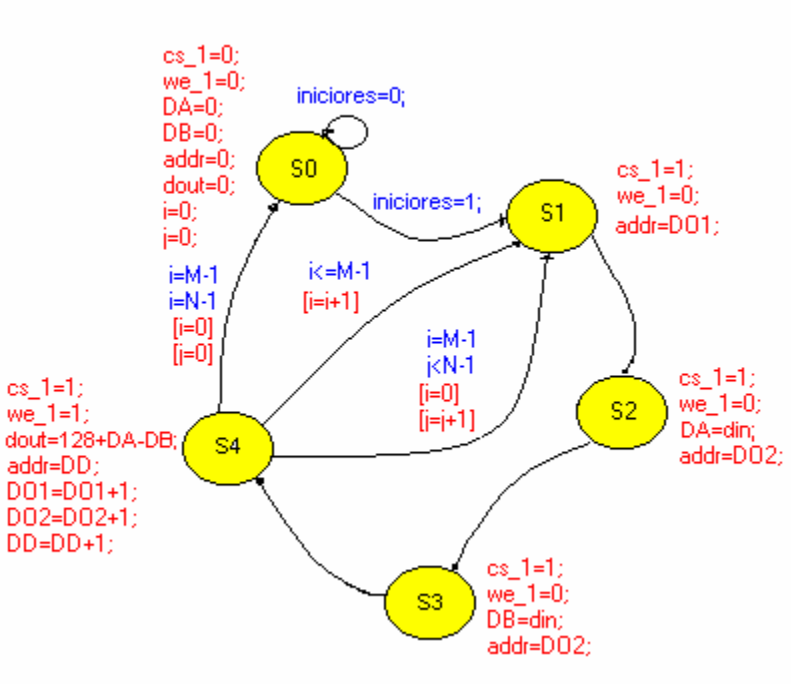


Figura 3.27. Diagrama de estados del restador de imágenes.

En el diagrama de estados de la figura anterior se detalla la resta de dos píxeles que se encuentran en la misma posición dentro de imágenes distintas dando lugar a otro píxel colocado en la misma posición que aquellos, pero en la imagen diferencia. Lo vemos estado a estado:

- ***S0***: El primer estado se corresponde con el estado de reposo. En él es importante que las salidas que van a parar a una puerta OR de entrada a la memoria de doble puerto, es decir, ADDR, CS_1 y WE_1 (ver esquema interno del coprocesador, figura 3.17), tengan un valor igual a cero. Permaneceremos en S0 hasta que Aquarius escriba un 1 en INICIORES.

- ***S1***: En este estado se activa la línea de habilitación de la memoria de doble puerto, tomando el restador de imágenes control sobre la misma. Se direcciona el primer elemento con la línea ADDR.

- ***S2***: En este estado el valor del primer elemento aparece por el puerto DIN, siendo almacenado en DA, se realiza una nueva operación de lectura de la DPRAM, direccionando el segundo elemento de nuevo con ADDR.

- ***S3***: En este estado el valor del segundo elemento aparece por DIN y es almacenado en DB.

- ***S4***: En este estado se escribe el dato resultante de la resta de los elementos (128 + DA - DB) en la dirección señalada por ADDR, teniendo en cuenta que es una resta especial que satura a los valores 0 ó 255, se actualizan los cursores i y j, que llevan el control de la posición del píxel a procesar dentro de la imagen y, según el valor de éstos, el ciclo que sigue será S0 o S1. En esta ocasión existen dos transiciones posibles de S4 a S1, ya que la actualización de los cursores dependerá de si nos encontramos en un elemento correspondiente a la última columna de una imagen o no. En el caso de que nos encontremos en el último píxel de la imagen, el estado siguiente sería S0, volviendo el sistema al reposo inicial.

En esta ocasión la resta se realiza sobre todos los píxeles de las imágenes a restar, de ahí que los valores de los cursores vayan desde 0 a M-1 (cursor i) y desde 0 a N-1 (cursor j).

3.3.5 Umbralizador

Este bloque es otro de los subsistemas del coprocesador visual que se encarga de generar una imagen del mismo tamaño que la original cuyos elementos tendrán el valor máximo (255), si superan cierto umbral (T), o el valor mínimo (0) si no lo superan. La imagen resultante será, por tanto, una imagen binaria, cuyos píxeles serán blancos (nivel de gris 255) o negros (nivel de gris 0).

Este bloque es independiente de los vistos anteriormente y sólo se comunica con la memoria de doble puerto y con Aquarius.

Esquema del umbralizador

La siguiente figura muestra un esquema del umbralizador:

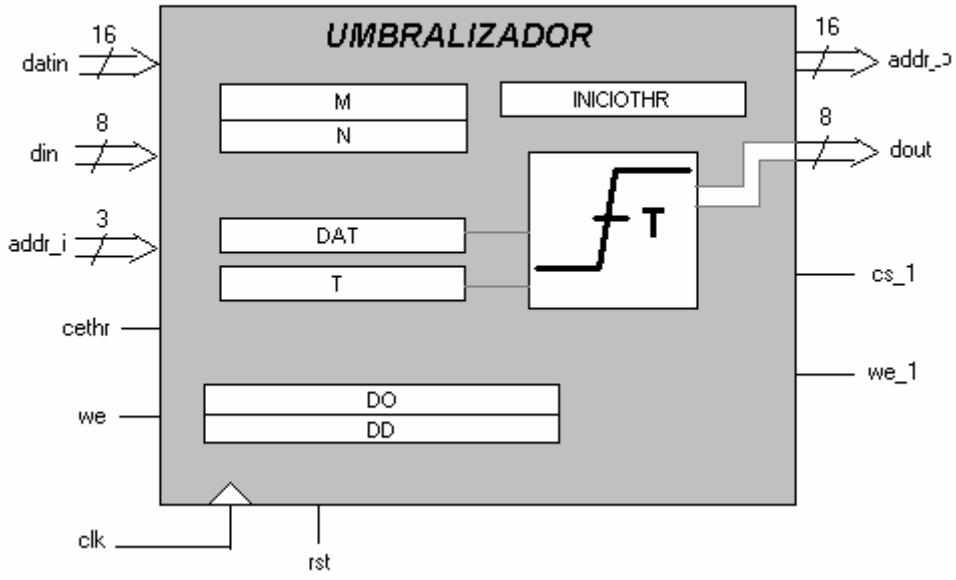


Figura 3.28. Esquema del umbralizador.

Estudiamos sus puertos de entrada / salida:

- **DATIN**: Línea de datos procedente de Aquarius dedicada a la configuración de los registros internos del sumador por parte de aquél.
- **DIN**: Línea de entrada de datos procedentes de la memoria de doble puerto.
- **ADDRESS**: Entrada de direcciones desde Aquarius. Su ancho es de 3 bits y se corresponden con los 3 bits menos significativos del bus de direcciones del procesador.

- **CETHR**: Señal de habilitación del sumador de imágenes. Procede de Aquarius y se activa cundo éste quiere programar los registros internos del subsistema.
- **WE**: Señal que indica si el ciclo de acceso al subsistema en de escritura (1) o de lectura(0). Procede de Aquarius.
- **CLK**: Reloj del sistema.
- **RST**: Reset del sistema.
- **ADDR_O**: Línea de direcciones de 16 bits de salida. A través de ella, el umbralizador indica la dirección de la memoria de doble puerto en la que va a escribir o de la que va a leer un dato.
- **DOUT**: Línea de datos de 8 bits de salida. Su destino es la memoria de doble puerto, donde se almacenará la imagen binaria.
- **CS_1**: Señal de habilitación de la memoria de doble puerto (DPRAM). Es una señal de salida.
- **WE_1**: Señal de salida que indica si el ciclo de acceso a la DPRAM es de escritura (1) o de lectura (0).

Estructura interna del umbralizador

El umbralizador posee unos registros configurables por parte de Aquarius que se detallan a continuación:

[THR] Address=0xABCD0500 R/W M

M7	M6	M5	M4	M3	M2	M1	M0

[THR] Address=0xABCD0501 R/W N

N7	N6	N5	N4	N3	N2	N1	N0

[THR] Address=0xABCD0502 R/W DO (Dir. Origen)

DO15	DO14	DO13	DO12	DO11	DO10	DO9	DO8
DO7	DO6	DO5	DO4	DO3	DO2	DO1	DO0

[THR] Address=0xABCD0504 R/W DD (Direcc. Destino)

DD15	DD14	DD13	DD12	DD11	DD10	DD9	DD8
DD7	DD6	DD5	DD4	DD3	DD2	DD1	DD0

[THR] Address=0xABCD0506 R/W INICIOTHR

Res.	Res.	Res.	Res.	Res.	Res.	Res.	Inic.

[THR] Address=0xABCD0508 R/W T

T7	T6	T5	T4	T3	T2	T1	T0

Tabla 3.19. Registros internos del umbralizador.

Detallamos cada uno de estos registros:

- **M**: Una de las dimensiones de la imagen. Aquarius es quien escribe su valor.
- **N**: La otra dimensión de la imagen. Aquarius la escribe.
- **DO**: Dirección de 16 bits de la memoria de doble puerto que indica el comienzo de la imagen original. Es un valor que introduce Aquarius.
- **DD**: Dirección de 16 bits de la memoria de doble puerto en donde se va a escribir la imagen binaria.

- **INICIOTHR**: Señal que activa Aquarius para que el umbralizador comience a operar.
- **T**: Valor umbral comprendido entre 0 y 255.

Funcionamiento del umbralizador

Como comentamos al introducir este subsistema, este bloque se dedica a recorrer una imagen original, leyendo los valores del nivel de gris de sus píxeles, generando, a su vez, una imagen binaria sujeta a los valores de niveles de gris leídos de la imagen original. Si para una posición dada, lee un valor de nivel de gris superior o igual a T (programable por software), la imagen resultante tendrá un píxel con nivel de gris 255 en esa posición. De lo contrario, tendrá un píxel con nivel de gris 0 en esa posición.

Una aplicación interesante de este subsistema es su utilización en el campo de la segmentación por niveles de gris de imágenes, donde el umbralizado (*thresholding*) es una de las técnicas más importantes.

Para que el umbralizador se ponga en funcionamiento es necesaria una previa configuración de los registros internos de éste por parte de Aquarius. Una vez ocurra esto, el sistema estará preparado para operar, y será lanzado por el procesador cuando escriba un 1 en el registro INICIOTHR. El comportamiento de este subsistema puede verse mejor observando su diagrama de estados:

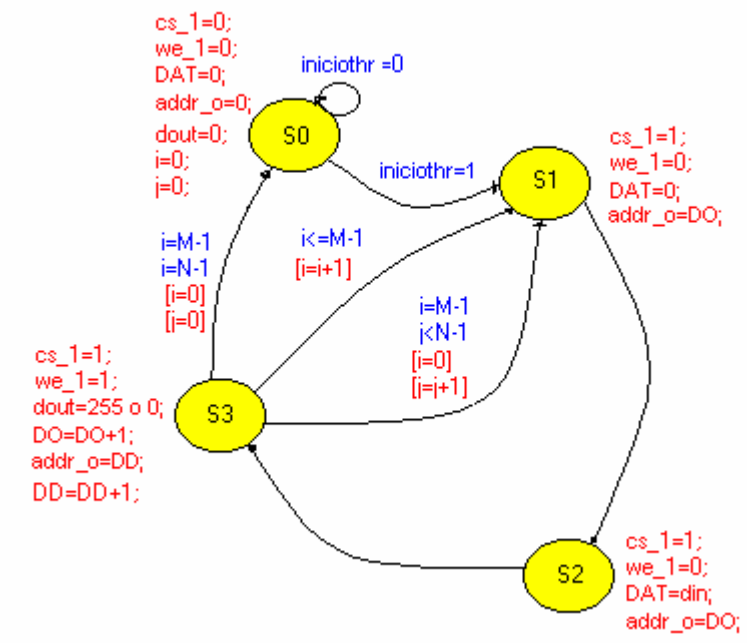


Figura 3.29. Diagrama de estados del umbralizador.

En el diagrama de estados de la figura anterior se detalla la comparación del valor del nivel de gris de un píxel con un umbral y la escritura de un 0 o un 255 en el valor del píxel situado en la misma posición de la imagen generada. Lo vemos estado a estado:

- ***S0***: El primer estado se corresponde con el estado de reposo. En él es importante que las salidas que van a parar a una puerta OR de entrada a la memoria de doble puerto, es decir, ADDR_O, CS_1 y WE_1 (ver esquema

interno del coprocesador, figura 3.17), tengan un valor igual a cero. Permaneceremos en S0 hasta que Aquarius escriba un 1 en INICIOTHR.

- ***S1***: En este estado se activa la línea de habilitación de la memoria de doble puerto, tomando el umbralizador control sobre la misma. La línea ADDR_O durecciona la DPRAM en busca del valor del píxel a umbralizar.

- ***S2***: En este estado se lee el valor del nivel de gris del píxel que se encuentra en la posición que indica aparece en la entrada DIN, almacenándose en el registro DAT.

- ***S3***: En este estado se escribe el dato resultante del umbralizado en la dirección señalada por ADDR_O, se actualizan los cursores i y j, que llevan el control de la posición del píxel a procesar dentro de la imagen y, según el valor de éstos, el ciclo que sigue será S0 o S1. De nuevo existen dos transiciones posibles de S3 a S1, ya que la actualización de los cursores dependerá de si nos encontramos en un elemento correspondiente a la última columna de una imagen o no.

En el caso de que nos encontremos en el último píxel de la imagen, el estado siguiente sería S0, volviendo el sistema al reposo inicial.

En esta ocasión el umbralizado se realiza sobre todos los píxeles de la imagen original, de ahí que los valores de los cursores vayan desde 0 a M-1 (cursor i) y desde 0 a N-1 (cursor j).

3.3.6 Memoria de doble puerto

Ya explicamos la necesidad de una memoria de este tipo en el apartado *3.2.3*, en el que vimos cómo procesador y coprocesador compartían una zona común de lectura y escritura de datos.

En esa zona es donde el procesador iba a almacenar la imagen recibida, y el coprocesador iba a actuar sobre ella, guardando el resultado del proceso en dicha zona.

Como ha ocurrido en el resto de apartados, al hablar de esta memoria lo haremos por su nombre o por la abreviatura DPRAM (acrónimo de Dual-Port RAM).

Esquema de la memoria RAM de doble puerto

La siguiente figura muestra los puertos de entrada y salida de la DPRAM:

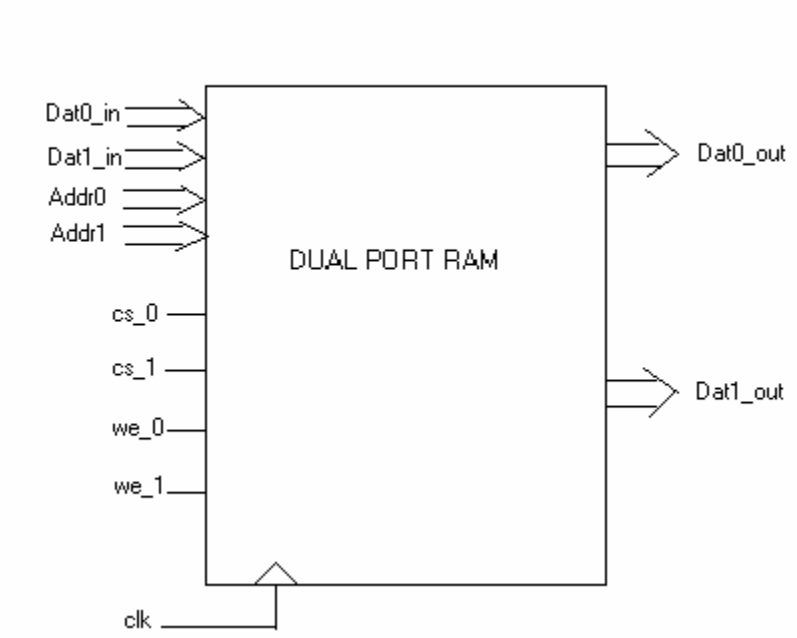


Figura 3.30. Interfaz de E/S de la memoria RAM de doble puerto.

Detallamos cada uno de los puertos:

- **DAT0_IN**: Línea de datos de entrada por el puerto 0 de la memoria. Este puerto es por el que accede Aquarius a depositar la imagen a tratar. En cada posición almacena el valor del nivel de gris de un píxel de la imagen. Es de 8 bits de ancho, con lo que esos valores estarán incluidos en el intervalo [0,255].
- **DAT1_IN**: Línea de datos de entrada por el puerto 1 de la memoria. Este puerto es por el que accede cada uno de los subsistemas del coprocesador. Sus características son idénticas a las del puerto 0.
- **ADDR0**: Línea de direcciones de entrada al puerto 0 de la memoria. Estas direcciones las pone Aquarius. Es de 16 bits de ancho, lo que permite que haya hasta 64Kbytes de memoria de datos de imágenes.
- **ADDR1**: Línea de direcciones de entrada al puerto 1 de la memoria. Son direcciones puestas por cada uno de los subsistemas de los que se compone el coprocesador. Su ancho es de 16 bits.
- **CS_0**: Señal de habilitación del puerto 0.
- **CS_1**: Señal de habilitación del puerto 1.
- **WE_0**: Señal que indica si un acceso al puerto 0 de la memoria se realiza en lectura (0) o en escritura (1). Se corresponde con la señal WE de Aquarius.
- **WE_1**: Señal que indica si un acceso al puerto 1 de la memoria se realiza en lectura (0) o en escritura (1). Está controlada por el coprocesador (por sus subsistemas).
- **CLK**: Reloj del sistema. Puede haber 2 relojes, uno para cada puerto, como explicaremos a continuación, pero estarían conectados ambos al reloj que comanda todo el sistema.

- **DAT0_OUT**: Línea de datos de salida por el puerto 0 de la memoria. Se corresponde con 8 bits del bus de datos de Aquarius, y su posición dentro del bus viene dada por la señal SEL, también perteneciente al procesador, como explicamos en el apartado 3.3.2.
- **DAT1_OUT**: Línea de datos de salida por el puerto 1 de la memoria. Se corresponde con una de las entradas de datos del MAC, con el puerto DIN del sumador de imágenes, o con el puerto DIN del umbralizador, dependiendo del subsistema que controle la memoria.

Memoria de simulación vs. Memoria implementada.

En este punto es importante hablar de área consumida, ya que los dispositivos reconfigurables en los que pueden ser implementados sistemas de este tipo, como son FPGAs, PLDs, etc. poseen recursos limitados y para aplicaciones de procesado de imagen existe un gran consumo de memoria debido al tamaño de las imágenes. Hay que tener en cuenta que una imagen de 32x32, considerada pequeña desde un punto de vista de su definición, va a ocupar 1KByte de espacio en memoria, teniendo en cuenta que cada píxel va a tener un valor de nivel de gris comprendido entre 0 y 255, es decir, va a estar representado por 8 bits. Una imagen de 64x64 ocuparía 4 KBytes.

También hay que pensar que no vamos a tener en memoria solamente la imagen a procesar, sino que las imágenes generadas como fruto del tratamiento de la imagen original, y que tienen el mismo tamaño que aquella, deben ser almacenadas igualmente, ya que, en algunos casos, pueden tratarse de imágenes intermedias necesarias para conseguir una imagen final deseada. Ejemplo de esto es el cálculo del gradiente, donde hay que generar sus componentes horizontal y vertical antes de obtener la imagen gradiente.

Tenemos un compromiso, por tanto, entre el área limitada de los dispositivos programables y la necesidad de gran almacenaje de las aplicaciones de procesado de imagen. Dicho esto, resulta clave encontrar un modo de definir la memoria de doble puerto que garantice un uso óptimo de los recursos de la FPGA en la que vamos a implementar el sistema. Ese método es el uso de BlockRAM para el desarrollo de la misma.

Algunas FPGAs poseen determinadas regiones destinadas a almacenar dispositivos de memoria. Estas zonas se encuentran formadas por bloques de una determinada anchura y longitud que están optimizados para albergar memorias, tanto de puertos simples como dobles. Esos bloques de los que están formados esas regiones reciben el nombre de BlockRAMs o BRAMs. Para ilustrar esta idea podemos observar la figura que se muestra a continuación:

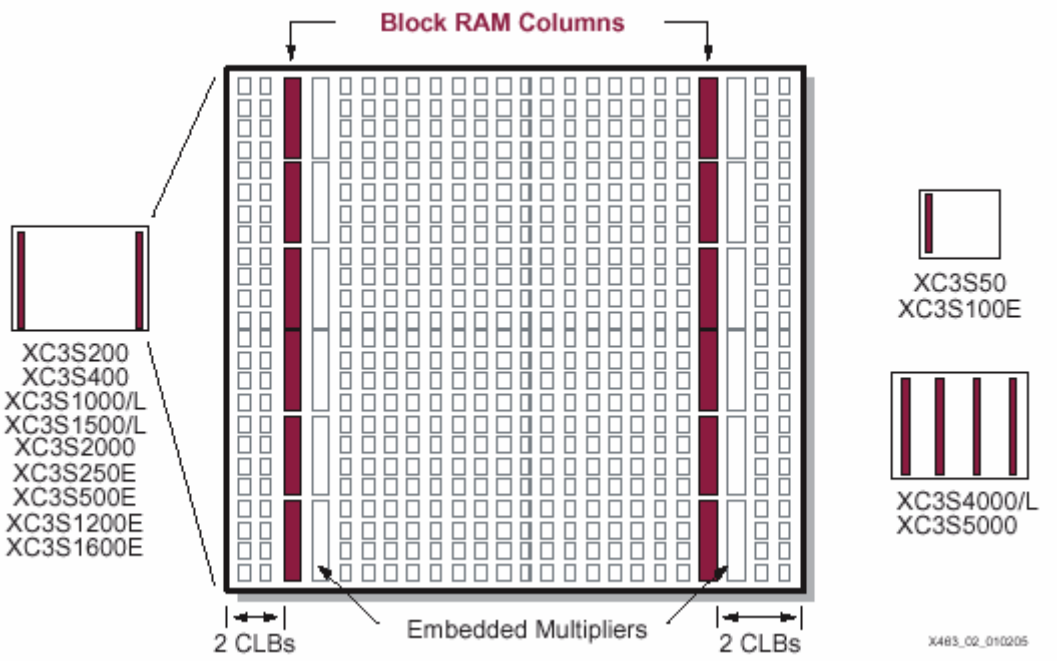


Figura 3.31. Representación gráfica del contenido de una FPGA. [27]

Un método sencillo de generar la memoria que deseemos utilizar y que esté definida mediante BRAMs es usar la herramienta CoregenIP contenida en el programa que hemos venido usando para el desarrollo del sistema, el Xilinx ISE v8.1.

Este programa es bastante fácil de entender por el usuario y tan sólo hay que indicar el tipo de memoria (RAM de doble puerto en nuestro caso) y las dimensiones de la misma. Vemos una muestra en la siguiente figura:

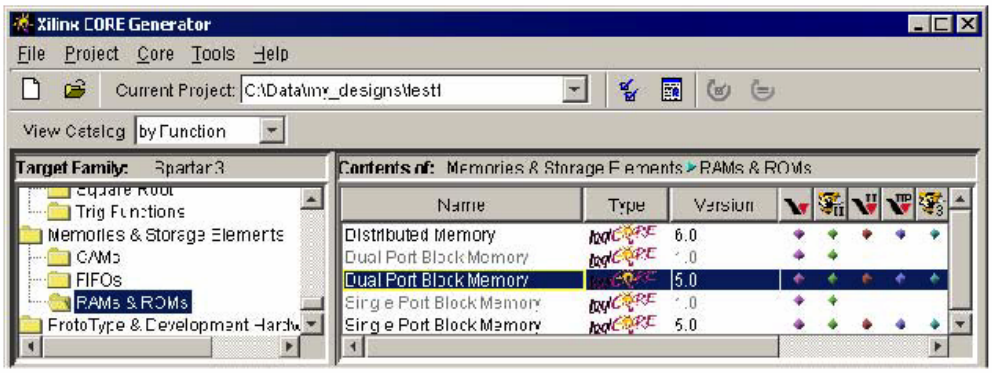


Figura 3.32. Generación de una memoria RAM de doble puerto usando Coregen IP.

Las dimensiones de la memoria serán:

- 8 bits de ancho: Para contener el valor del nivel de cada píxel.
- 65536 direcciones: 16 bits de bus de direcciones para ambos puertos. Esta cantidad representa una capacidad razonable para almacenar un número limitado de imágenes de un tamaño también acotado. Por ejemplo, podría almacenar 16 imágenes de tamaño 64x64 píxeles o 64 imágenes de 32x32 píxeles, etc.

Esta memoria que hemos definido es imprescindible para llevar a cabo una correcta implementación, y es la que usaremos para ese fin. El problema es que el programa nos impide simular elementos generados por la herramienta Coregen IP, por lo que debemos usar otra, definida como queramos, que respete los puertos de la generada. En nuestro caso particular la memoria que hemos usado para simulación posee la estructura externa mostrada en la figura 3.30, que se diferencia de la generada por Coregen IP en que posee un único reloj, mientras que la generada tiene un reloj para cada puerto. En la práctica es algo que no va a afectarnos, ya que existe una única señal de reloj para todo el sistema.

4. IMPLEMENTACIÓN DEL SISTEMA SOBRE UNA FPGA

En este apartado vamos a describir el procedimiento seguido para la implementación física del sistema de visión, así como el conjunto de herramientas usadas para el diseño, depuración y comunicación entre los distintos componentes del entorno de desarrollo.

4.1 Descripción del entorno de desarrollo

Para verificar que el sistema de visión opera correctamente vamos a realizar un montaje tal como el mostrado en la figura 3.1, es decir, con un PC que se comunica, mediante el puerto serie, con el sistema de procesado, el cual se halla implementado en una FPGA.

Pasamos a describir cada uno de los componentes del entorno de desarrollo:

- **PC**: Ordenador personal con sistema operativo Windows XP que cuenta con las siguientes herramientas software: Xilinx ISE, v.8.1, Modelsim Starter Edition, v.5.5, Visual Basic .Net Express Edition, Matlab 6.5.
- **Cable RS232 y conectores:** En la conexión PC – FPGA para el intercambio de datos.
- **Cable USB y conectores:** En la conexión PC – FPGA para la configuración de la misma.
- **FPGA:** Dispositivo programable en cuyo interior se implementan físicamente Aquarius, el coprocesador y la memoria de doble puerto para almacenamiento de imágenes.
- **Tarjeta de test:** Contiene la FPGA, así como una amplia gama de recursos destinados a dotar a la FPGA de facilidades para su comunicación con una gran variedad de dispositivos que siguen distintas interfaces, permitiendo el desarrollo de una multitud de aplicaciones diferentes.

Describimos las funciones de las distintas herramientas software utilizadas:

- **XILINX ISE:** Permite el diseño de sistemas utilizando lenguajes de descripción hardware (Verilog/VHDL), esquemáticos, diagramas de estado y otras utilidades. También se usa para la síntesis e implementación de los diseños realizados, así como la configuración y programación de la FPGA a utilizar.
- **MODELSIM:** Usado para la simulación de diseños. Permite, junto con Xilinx ISE, realizar simulaciones post-implementación, lo que permite depurar el sistema diseñado teniendo en cuenta factores propios de la colocación y rutado dentro de la FPGA, como retrasos de la señal de reloj, transitorios de señales, etc., permitiendo ver cómo se comporta el diseño tal y como se va a implementar de manera física.

- **Visual Basic:** Encargado de la interfaz entre el PC y la FPGA con el sistema en funcionamiento, a través del puerto serie. Explicado con más detalle en el apartado 4.3.
- **Matlab:** Usado para adecuar una imagen al formato utilizado por la FPGA.

4.2 Tarjeta de test y desarrollo

La tarjeta de test y desarrollo que vamos a utilizar es la XUP-V2P [28], que posee una FPGA Virtex II Pro XC2VP30. La tarjeta posee circuitos de configuración por JTAG, USB2, y FLASH ROM, así como circuito de alimentación e interfaces diversas para comunicación con otros dispositivos.

La siguiente figura muestra la tarjeta con sus distintos componentes:

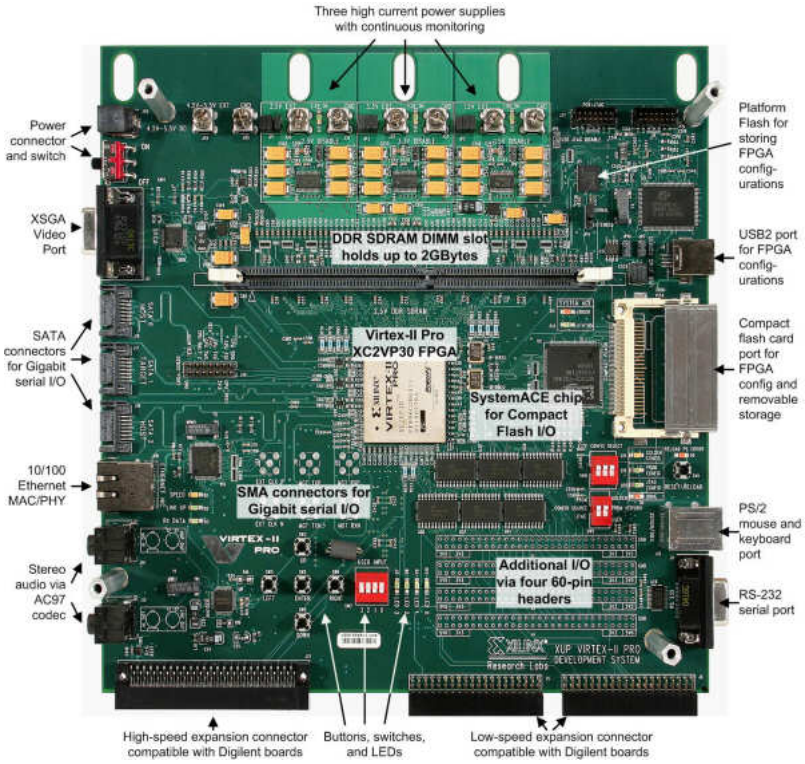


Figura 4.1. Tarjeta de desarrollo XUP V2P [28]

Dentro de las características de la placa XUP V2P destacamos:

- Posee una FPGA Virtex II Pro con dos procesadores Power PC en su interior.
- Hasta 2 GB de SDRAM.
- Memoria Flash y sistema ACE para configuración de la FPGA y almacenamiento de datos.
- Puerto USB para configuración.
- Interfaz Ethernet 10/100.
- Puerto serie RS232 DB9.
- Dos puertos serie PS2.
- 4 LEDs conectados a pines de entrada / salida de la FPGA.

- 5 botones de pulsado conectados a pines de entrada / salida de la FPGA.
- 4 conmutadores conectados a pines de entrada / salida de la FPGA.
- Reloj de 100 MHz.
- Provisión para relojes suministrados por el usuario.
- Circuitos de alimentación "on-board", etc.

La tarjeta XUP V2P está diseñada de tal modo que admite dos FPGA's posibles, la XC2VP20 y la XC2VP30, siendo ésta última la usada por nosotros. Veamos las características de ambas en la siguiente tabla:

Característica	**XC2VP20**	**XC2VP30**
Slices	9280	13969
Tamaño de Array	56 x 46	80 x 46
RAM distribuída	290kb	428kb
Bloques multiplicadores	88	136
Block RAMs	1584kb	2448kb
DCMs (Digital Clock Managers)	8	8
Power PC RISC cores	2	2
Multi-Gigabit Transceivers	8	8

Tabla 4.1. Tabla comparativa de FPGA XC2VP20 y FPGA XC2VP30.

Una vez descrita la tarjeta de desarrollo y la FPGA a utilizar, vamos a tratar de aprovechar al máximo la velocidad del sistema, para lo cual es importante analizar el resultado de la síntesis del sistema diseñado, lo que nos dará la frecuencia a la que vamos a trabajar. Una vez determinada la frecuencia, debemos revisar los cálculos hechos para determinar la tasa de transmisión y recepción de datos de la UART, según la expresión 3.1, que vimos en el apartado 3.2.1.b.

Síntesis del sistema

Para obtener la síntesis de nuestro diseño hemos de arrancar Xilinx ISE v 8.1 y abrir un nuevo proyecto. A continuación se abre un asistente para la creación de un nuevo proyecto. En esa ventana hemos de indicar el flujo de diseño que vamos a seguir, así como el dispositivo configurable que vamos a usar. En nuestro caso seguiremos un flujo de diseño ***XST Verilog*** (lenguaje de descripción hardware usado) y nuestro dispositivo es ***x2cvp30-7ff896***, valores obtenidos al examinar el exterior de la FPGA.

Una vez finalicemos la creación del proyecto (indicar ***Siguiente*** en el resto de opciones) hemos de incluir el sistema diseñado (es decir, los ficheros *verilog* del diseño) en el directorio correspondiente al proyecto generado. Entonces hay que añadir los ficheros *verilog* señalando en la barra de herramientas ***Project->Add source.*** Hemos de introducir en el proyecto los siguientes archivos:

top.v	defines.v	timescale.v
cpu.v	datapath.v	register.v
mult.v	mem.v	memory_fpga.v
lib_fpga.v	sys.v	pio.v
uart.v	sasc_brg.v	sasc_top.v

sasc_fifo4.v	toptotal.v	decode.v
convol.v	sumador.v	mac.v
restador.v	threshold.v	

El siguiente paso es generar la memoria RAM de doble puerto, que ha de estar definida mediante BlockRAMs, como vimos en el apartado 3.3.6. Para ello seleccionamos ***Project->New source***. Marcamos CoreGen IP y llamamos al fichero a obtener *dpram.v*, para que se incluya correctamente a la jerarquía del diseño. Seguimos los pasos indicados en aquél apartado.

Ahora ya tenemos el sistema completo. Estamos en condiciones de sintetizar el proyecto. Marcamos con el puntero el bloque *top.v*. Entonces buscamos la ventana de procesos, colocada debajo de la que contiene la estructura jerárquica del diseño. Una vez en ella, colocamos el puntero sobre ***Synthesize***, señalamos con el botón derecho y pulsamos ***Run.***

Al finalizar el proceso podemos observar el fichero con los resultados de la síntesis, en el que se indica los porcentajes de ocupación de los distintos bloques dentro de la FPGA. En el fichero vemos que la frecuencia máxima de funcionamiento es de unos 51 MHz, con lo que hemos de retocar nuestro diseño para que soporte la frecuencia de reloj que le va a hacer funcionar, ya que ésta es de 100MHz. Es por esa razón por la que hay que añadir el fichero *divide_clk.v*, que se encarga de reducir la frecuencia de reloj a la mitad, 50MHz. Por tanto hay que corregir los cálculos que afectaban a las velocidades de transmisión y recepción de la UART, usando la fórmula dada por la expresión 3.1, y obteniendo los siguientes resultados:

Tasa Baudios [bps]	f (CLK) [MHz]	UARTBG0	UARTBG1	Notas
1200	50	0x3C (60)	0xA8 (168)	
2400	50	0x3C (60)	0x54 (84)	
4800	50	0x12 (18)	0x82 (130)	
9600	50	0x12 (18)	0x41 (65)	

Tabla 4.2. Tasa de baudios corregida para una frecuencia de 50MHz.

Una vez completada la síntesis, realizamos la conversión del programa en lenguaje C, que va a ser el que carguemos en la memoria interna de Aquarius, en sentencias del tipo INIT, que son una forma de inicialización de la memoria interna, definida como BlockRAM, y las añadimos al fichero de restricciones del usuario (*.ucf*), junto con la asignación de los pines de la FPGA que vamos a utilizar. Este proceso de conversión se realiza mediante la aplicación *genram.c*, de la que hablamos en el apartado de programación de aplicaciones en Aquarius.

Implementación del sistema

El siguiente paso es lanzar la implementación física, que consta de diversas fases, tales como el mapeado del diseño dentro de la FPGA, la colocación (*place*), y el rutado (*route*). Para ello, colocamos el puntero del ratón encima de ***Implement Design***, pulsamos el botón derecho y accionamos la orden ***Run***.

Configuración de la FPGA

El último paso es la configuración de la FPGA, para lo cual usaremos el programa iMPACT. Para lanzarlo, buscamos ***Configure Device (iMPACT)***, dentro de ***Generate Programming File*** (ver figura 4.2), marcamos con botón derecho y lanzamos ***Run***.

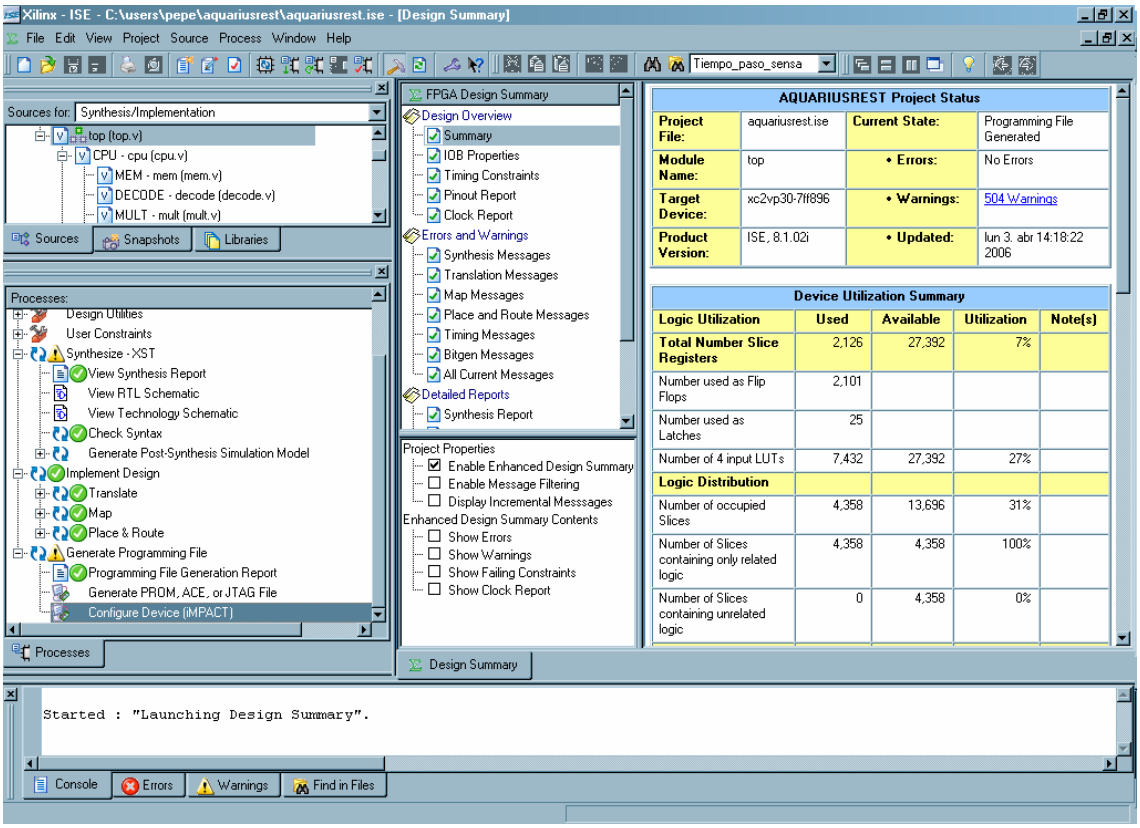


Figura 4.2. Configuración FPGA usando iMPACT.

Al finalizar el proceso, se abrirá una ventana correspondiente al programa iMPACT. Ahora es el momento de encender la placa y conectar el cable USB a la tarjeta y al PC. Una vez detectado el dispositivo, hemos de programar el tipo de configuración que vamos a realizar. Para ello buscamos en la barra de herramientas ***Output->Cable Setup***. En la ventana que se nos abre debemos marcar la opción correspondiente al cable USB y elegir una velocidad adecuada, por ejemplo, elegimos ***Select Speed*** y pulsamos ***OK***, y seleccionamos 6MHz. A continuación pulsamos el botón derecho del ratón y seleccionamos ***Initialize Chain***, que abre tres ventanas, cada una para cada uno de los bloques que aparecen en la figura 4.3. Para los dos primeros bloques marcamos ***Bypass***, ya que no son el objeto de nuestro proceso de configuración. Para el tercer bloque, correspondiente a la FPGA, marcamos el fichero *top.bit*, que es el código binario que vamos a enviar a la FPGA, y seleccionamos ***Aceptar***. Por último, colocamos el puntero del ratón sobre el bloque correspondiente a la FPGA, pulsamos el botón derecho del ratón y seleccionamos ***Program***.

Marcamos de nuevo *top.bit* en la nueva ventana emergente y pulsamos ***OK***. El resultado final es el mostrado en la siguiente figura.

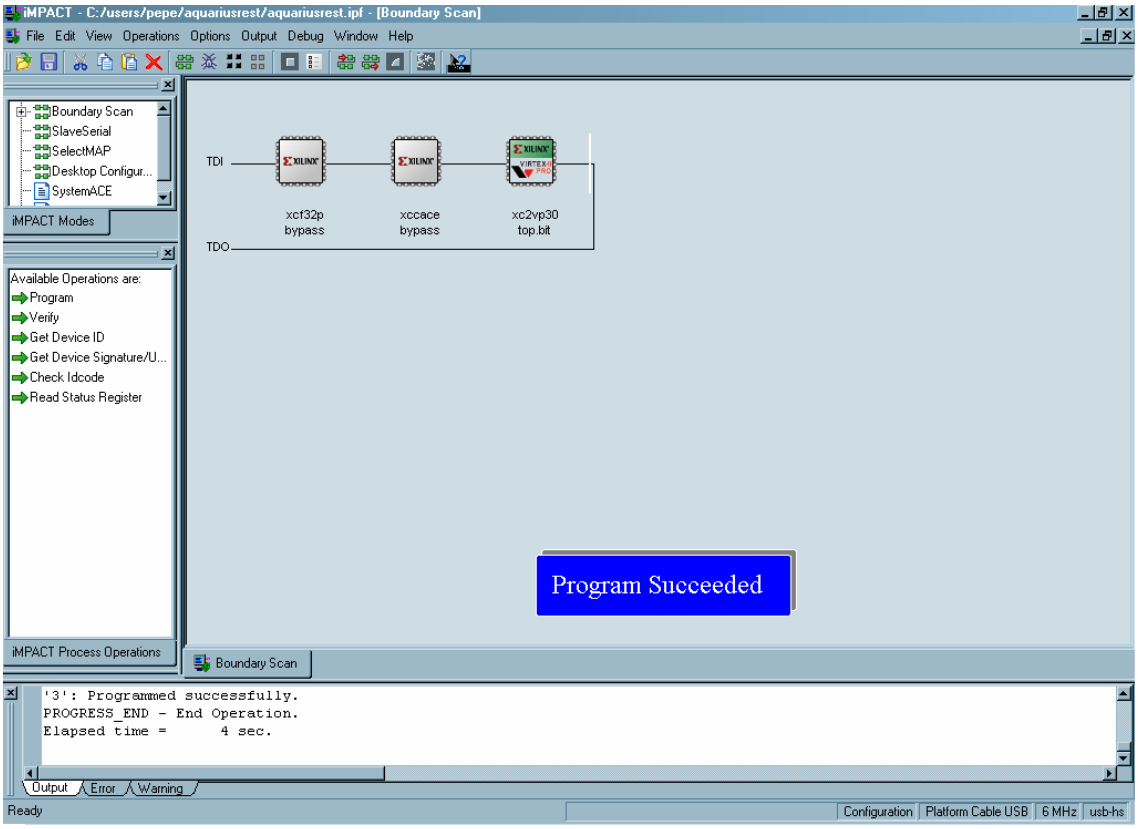


Figura 4.3. Pantalla final del proceso de configuración de la FPGA usando iMPACT

Una vez completado el proceso, el sistema se encuentra en funcionamiento realizando la tarea que se le haya programado.

4.3 Interfaz con el PC

Existen dos tipos de comunicación entre la tarjeta de desarrollo y el PC, el intercambio de información durante la ejecución del programa (a través del puerto serie RS232), y el envío de la información de configuración desde el PC a la tarjeta (por puerto USB, comentada ya en el apartado anterior).

Nos centramos en el primer caso.

Configuración de la comunicación

Para que la interacción del usuario con la tarjeta programada sea exitosa, hay que seleccionar una velocidad de transferencia adecuada, que sea común a ambos dispositivos (tarjeta y PC).

Es importante que ambos extremos de la comunicación usen el mismo protocolo para el puerto serie, ya que existen varios, entre ellos:

- **TxD, RxD**: El protocolo más sencillo usa sólo las señales de transmisión y recepción de datos serie.

- **TxD, RxD, CTS, RTS**: El mismo que el anterior, pero usando dos señales de protocolo más, CTS (Clear To Send) y RTS (Request To Send). Es el usado por defecto en Aquarius.

Vamos a usar el primero de los protocolos, por hacer más sencilla la interfaz desde el PC, y por hacer más rápido el intercambio de datos. Es por ello por lo que debemos modificar Aquarius para que se adapte al protocolo. Lo único que debemos hacer es cortocircuitar de manera interna la señal de salida de la unidad microcontroladora, RTS, con la otra señal a eliminar del protocolo, CTS, de entrada a Aquarius. Para ello, quitamos las mismas de la entidad de más alto nivel del sistema, *top.v*, y realizamos la siguiente asignación dentro del propio fichero verilog:

```
assign CTS = RTS;
```

De este modo, cuando la UART de la MCU realice una petición de envío (es decir, active RTS), obtendrá el permiso correspondiente al activarse CTS (preparado para transmitir).

Entorno usado por el PC

Para el intercambio de información con la tarjeta a través del puerto serie, podemos usar varias opciones, por ejemplo:

- **Hyperterminal**: Podemos usar una sesión de este tipo para comunicarnos con la tarjeta. El inconveniente es que envía caracteres de uno en uno, a través del teclado del PC, y en código ASCII, siendo poco útil a la hora de enviar/recibir datos de una imagen, por ejemplo, donde los caracteres son usados como números enteros representados por 8 bits, de valores entre 0 y 255.
- **Matlab**: Permite el envío y recepción de grupos de caracteres, así como una gran gama de opciones de configuración de la transferencia de datos. Parece una buena opción, por tanto.
- **Visual Basic**: Sus prestaciones son parecidas a las de Matlab, con la posibilidad de configurar múltiples aspectos de la comunicación, así como de la información transmitida y recibida. Existen versiones gratuitas de este software, por lo que es la mejor opción.

Usamos un programa que gestiona el puerto serie (en apéndice 8.3) permitiendo el envío y recepción de imágenes, así como la visualización de las mismas. Para ello hemos de indicar la imagen a procesar en formato texto, encargándose el programa de generar las imágenes resultado en dicho formato, además de hacerlas visibles en la propia ventana de ejecución. Valga como ejemplo esta captura de la ventana del programa:

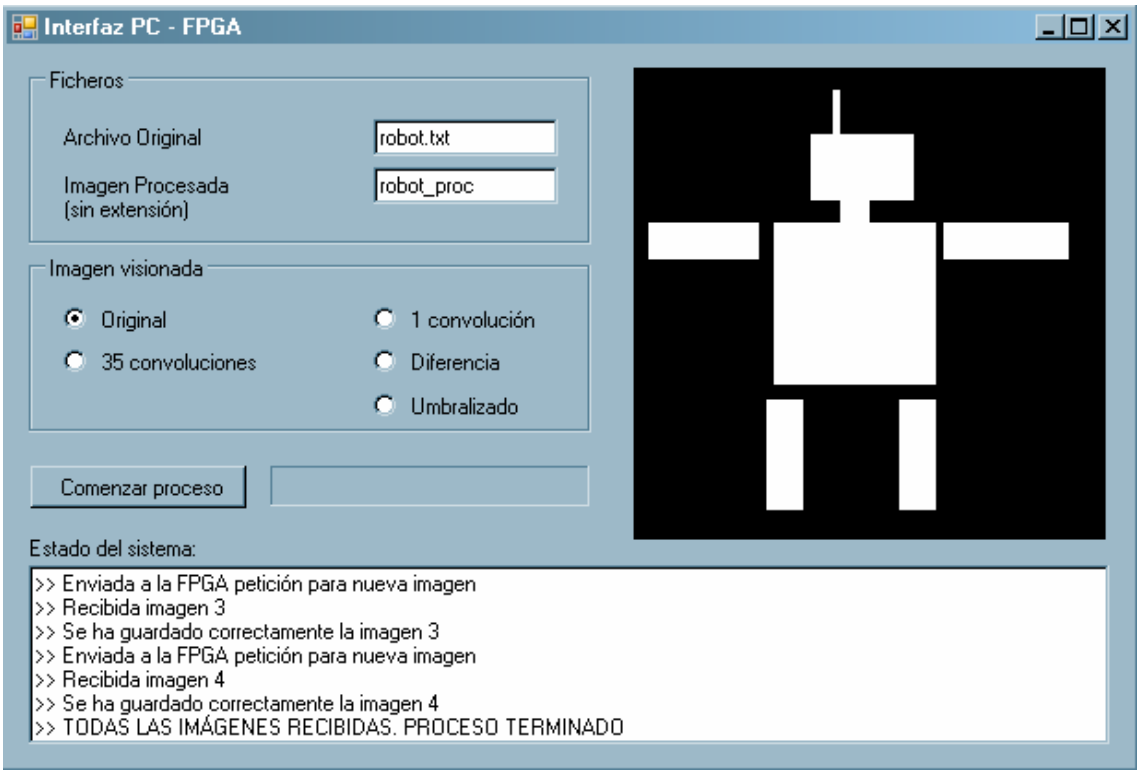


Figura 4.4. Diseño de la interfaz PC – FPGA, con imagen visualizada.

5. EJEMPLOS DE OPERACIÓN

5.1 El problema

El problema de procesamiento digital de imágenes que vamos a tratar como ejemplo es una simulación del comportamiento de una CNN (Cellular Neural Network), como vimos en el apartado 2.2.2. El objetivo de este ejemplo es la detección de bordes de una imagen en tonos de gris. El diagrama de flujo mostrado a continuación indica las operaciones a realizar sobre la imagen original:

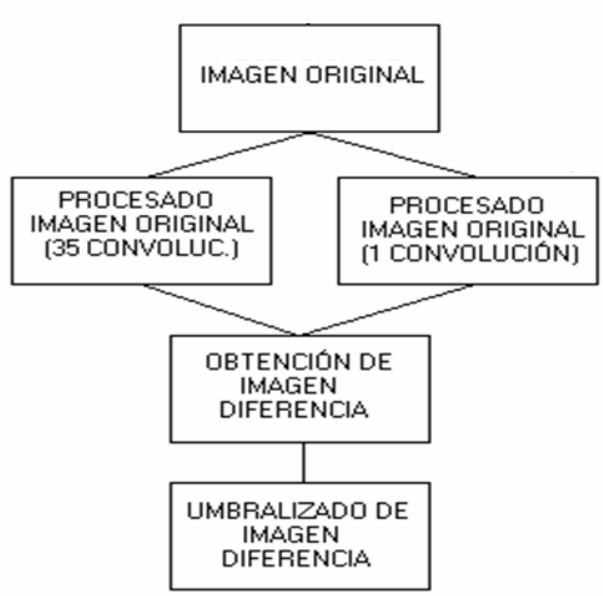


Figura 5.1. Diagrama de flujo del procesamiento de imagen

En primer lugar vamos a eliminar todas las pequeñas imperfecciones debidas al ruido en la captura y/o transmisión de la imagen y así como en su digitalización. Para ello, vamos a utilizar un filtro espacial de paso de baja, h(*x*,*y*) [29], ecuación (5.1) que se realiza mediante la difusión de los valores de los píxeles en la imagen original. La frecuencia[12] de corte espacial de dicho filtro paso de baja es inversamente proporcional al tiempo que dejamos evolucionar la difusión (Δt). *D* es la constante de difusión:

$$h(x, y) = \frac{1}{4\pi D \Delta t} e^{-\frac{(x^2+y^2)}{4 D \Delta t}} \qquad (5.1)$$

Puesto que en este sistema, la dinámica de la difusión está discretizada, no sólo en el espacio, sino también en el tiempo, vamos a controlar el tiempo de la difusión mediante el número de iteraciones de la máscara de difusión (ecuación (2.43)) sobre la imagen original. Estaremos realizando una simulación Forward-Euler de las ecuaciones diferenciales que describen la difusión en la red resistiva.

[12] Se trata de frecuencias espaciales.

A continuación se calcula la imagen diferencia entre dos versiones filtradas de la imagen original con diferente frecuencia de corte (f_{c1} y f_{c2}), esto se consigue realizando en un caso una sola convolución con la máscara de difusión (f_{c2}), y en otro varias (f_{c1}). Así, por un lado, los bordes que aparecen de manera espuria, y que corresponden por tanto a componentes de altas frecuencias espaciales, desaparezcan, y por otro lado, las componentes de menor frecuencia, que vienen a representar todo lo que no es un borde en la imagen, también. La siguiente figura muestra el filtrado paso de banda conseguido (gris oscuro), al realizar la diferencia entre las dos dinámicas (desaparece lo marcado en gris claro):

Figura 5.2. Filtrado realizado sobre la imagen.

Finalmente se realiza un umbralizado de esta diferencia para concluir con una imagen binaria que contenga esencialmente los píxeles que pertenecen a los bordes de los objetos que componen la escena.

Para realizar este procesamiento, en nuestro sistema de test, vamos a enviar una imagen desde el PC que será procesada por la FPGA, de tal modo que obtengamos dos versiones de la misma, dos difusiones de la imagen original con diferentes constantes de tiempo, es decir, vamos a realizar un número diferente de operaciones de convolución sobre la imagen original, generando una imagen resultado muy procesada (un número alto de convoluciones realizadas) y otra menos procesada (menos convoluciones), para simular el efecto de usar constantes de tiempo diferentes. Tras esto, llevaremos a cabo una resta de las versiones difundidas de la imagen original, obteniendo como resultado una imagen diferencia, que a su vez será sometida a una operación de umbralizado. El resultado será una imagen binaria, en la que aparecerán marcados los bordes de la imagen original.

El número de operaciones de convolución que vamos a realizar sobre la imagen recibida en nuestro ejemplo en concreto será de 35 y 1, respectivamente, para obtener cada una de las versiones procesadas con diferente constante de tiempo. La imagen a procesar será de tamaño 64x64, 4096 bytes, para aprovechar al máximo el buffer de datos enviados y recibidos por el puerto serie de Windows XP.

El diagrama de flujo del programa en C que vamos a escribir se muestra a continuación:

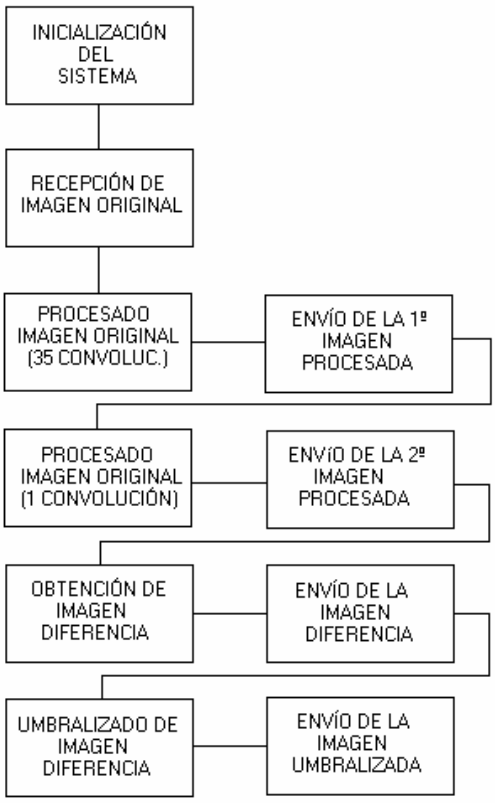


Figura 5.3. Diagrama de flujo del programa en C.

El programa que va a ser ejecutado en la FPGA, que puede verse en el apéndice 8.2, correspondiente a código C, incluye un protocolo sencillo con el programa diseñado en Visual Basic de tal forma que permita un correcto intercambio de las imágenes entre ambos. Este protocolo consiste en hacer que el sistema de visión espere la recepción de un dato, inútil para la ejecución del programa, después del envío de cada una de las imágenes procesadas, de manera que el PC pueda recibir una sola imagen procedente de la FPGA, evitando que el sistema de visión sobrescriba datos aún no leídos del buffer de entrada al PC.

5.2 Resultados

Después de llevar a cabo el proceso descrito en el apartado 4.2, tendremos programada la tarjeta de desarrollo, ejecutando la aplicación que hemos cargado en la memoria interna de Aquarius. En este caso, el sistema de visión va a esperar que le sea enviada una imagen desde el PC, para llevar a cabo una detección de borde sobre ella, simulando el comportamiento de una CNN.

Veamos qué imágenes recibiría el PC en cada uno de los pasos del diagrama de flujo de la figura 5.3, partiendo de cada una de las siguientes imágenes originales enviadas por el PC a la FPGA. La primera corresponde a una imagen real, con un bajo nivel de ruido debido a la captura y cuantización, procedente de un microscopio:

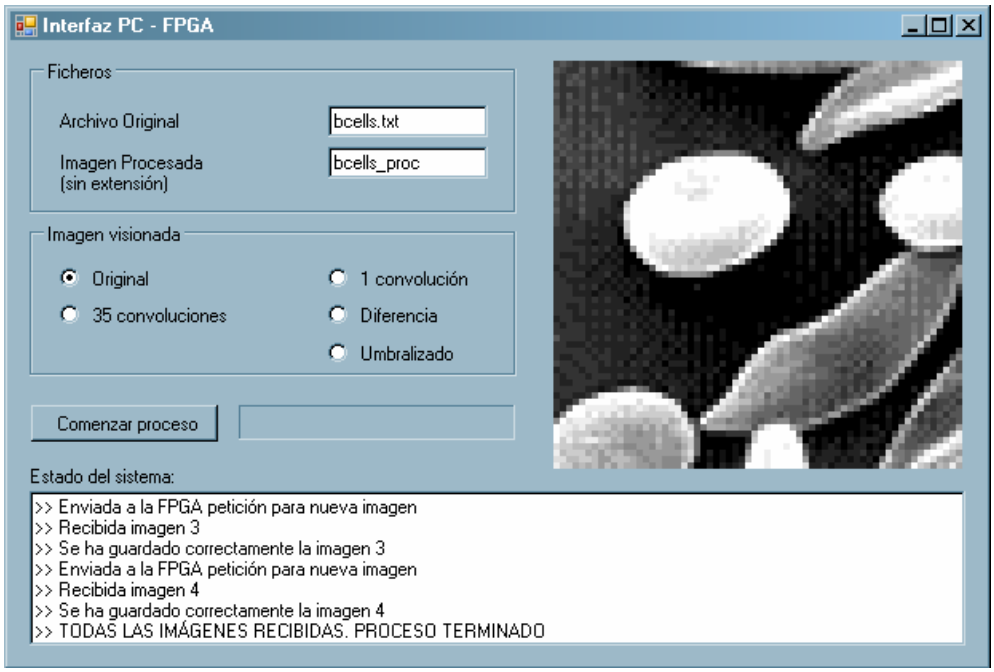
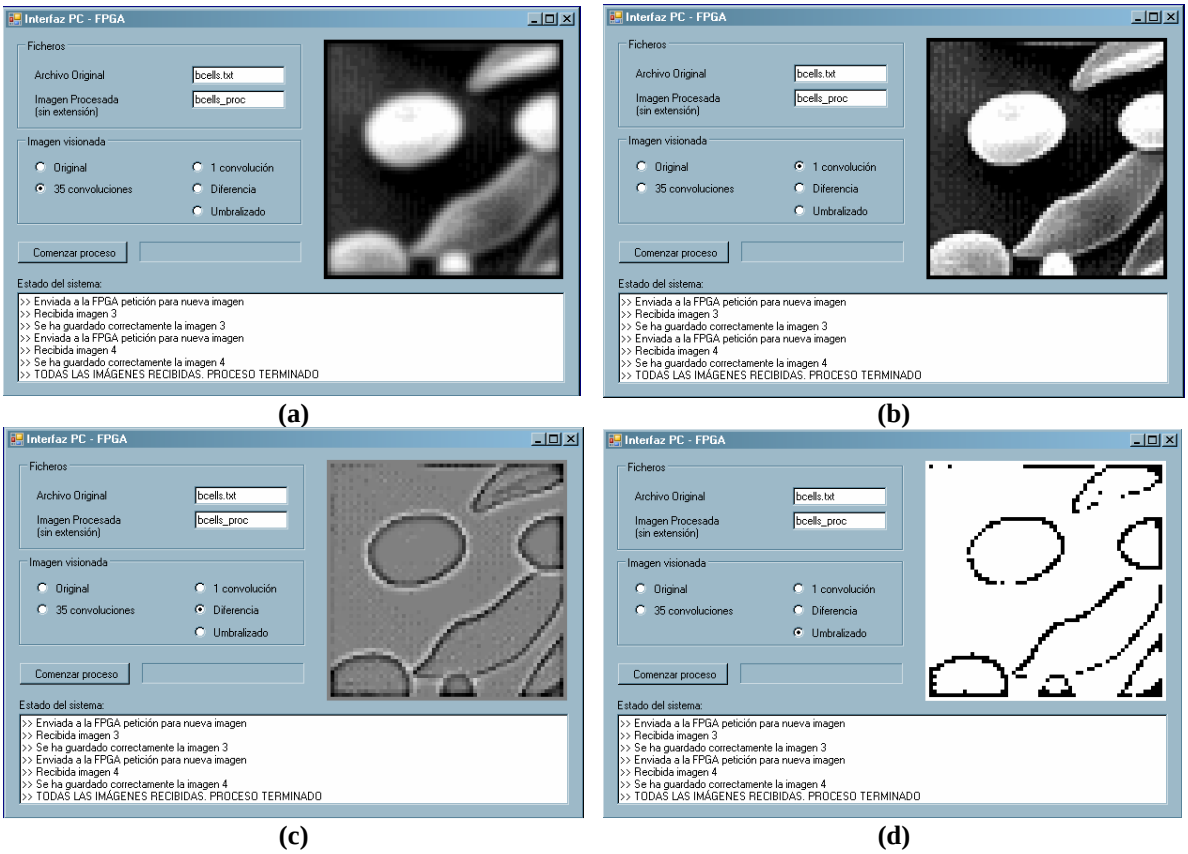


Figura 5.4. Imagen original, procedente de un microscopio, enviada a FPGA

Las imágenes resultantes del procesado:



(a) **(b)**

(c) **(d)**

Figura 5.5. Imágenes resultantes del procesamiento por parte del sistema de visión.

La figura anterior muestra las imágenes enviadas desde la FPGA al PC, como resultado de procesar la imagen de la figura 5.4:

(a) Imagen resultante de aplicar 35 convoluciones a la original.
(b) Imagen resultante de aplicar 1 convolución a la original.
(c) Imagen diferencia de las dos anteriores.
(d) Imagen umbralizada.

A continuación, enviamos a la FPGA la siguiente imagen original, esta vez se trata de una imagen sintética carente de ruido:

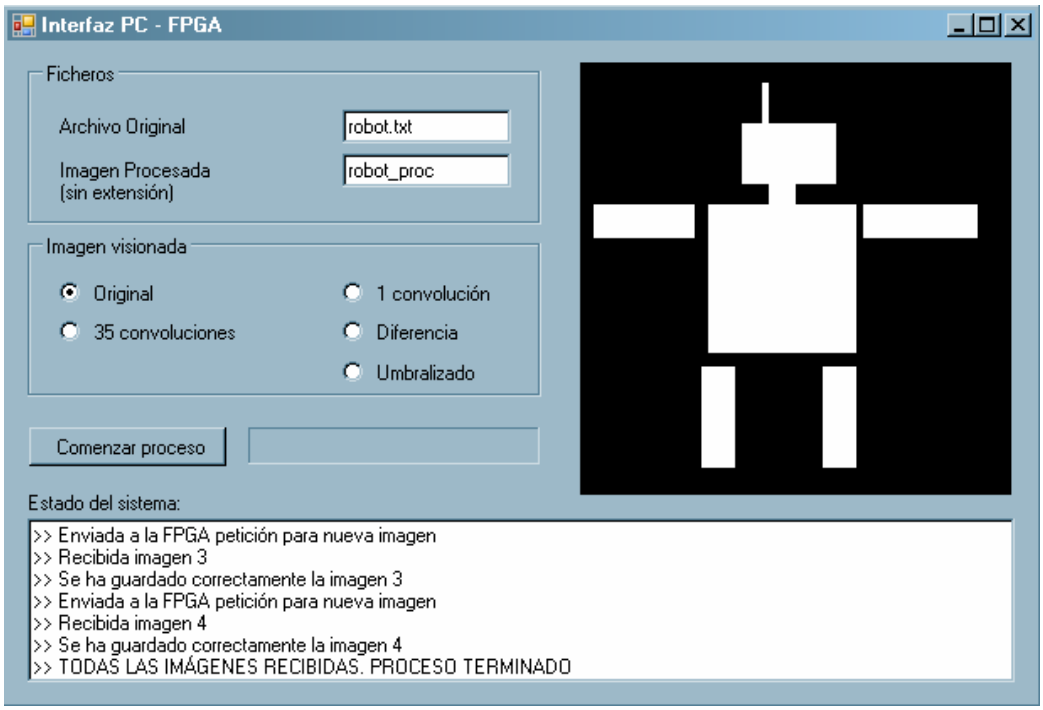
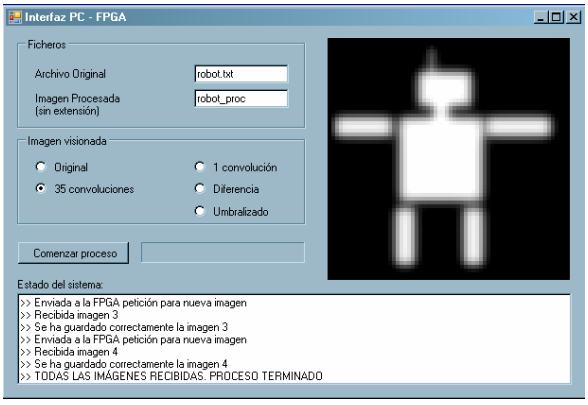
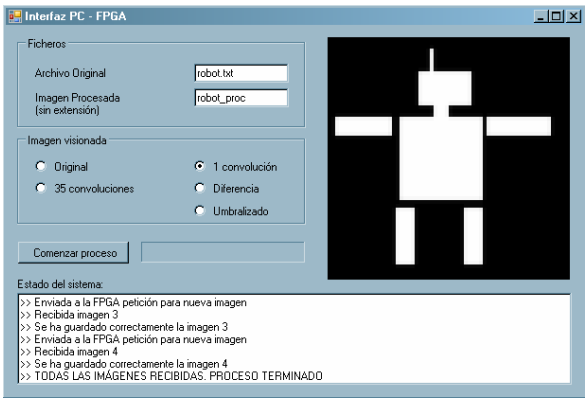


Figura 5.6. Imagen sintética enviada a la FPGA.

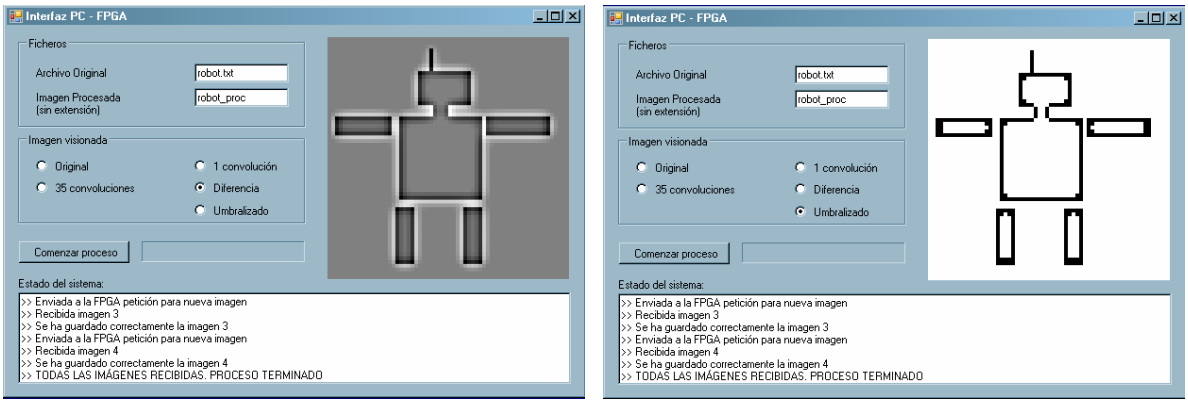
Las imágenes resultantes del procesado:



(a)



(b)



(c) **(d)**

Figura 5.7. Imágenes resultantes del procesamiento por parte del sistema de visión.

La figura anterior muestra las imágenes enviadas desde la FPGA al PC, como resultado de procesar la imagen de la figura 5.6:

(a) Imagen resultante de aplicar 35 convoluciones a la original.
(b) Imagen resultante de aplicar 1 convolución a la original.
(c) Imagen diferencia de las dos anteriores.
(d) Imagen umbralizada.

Enviamos, por último, una nueva imagen, esta vez con un efecto de la pixelización mucho más notable que en las anteriores:

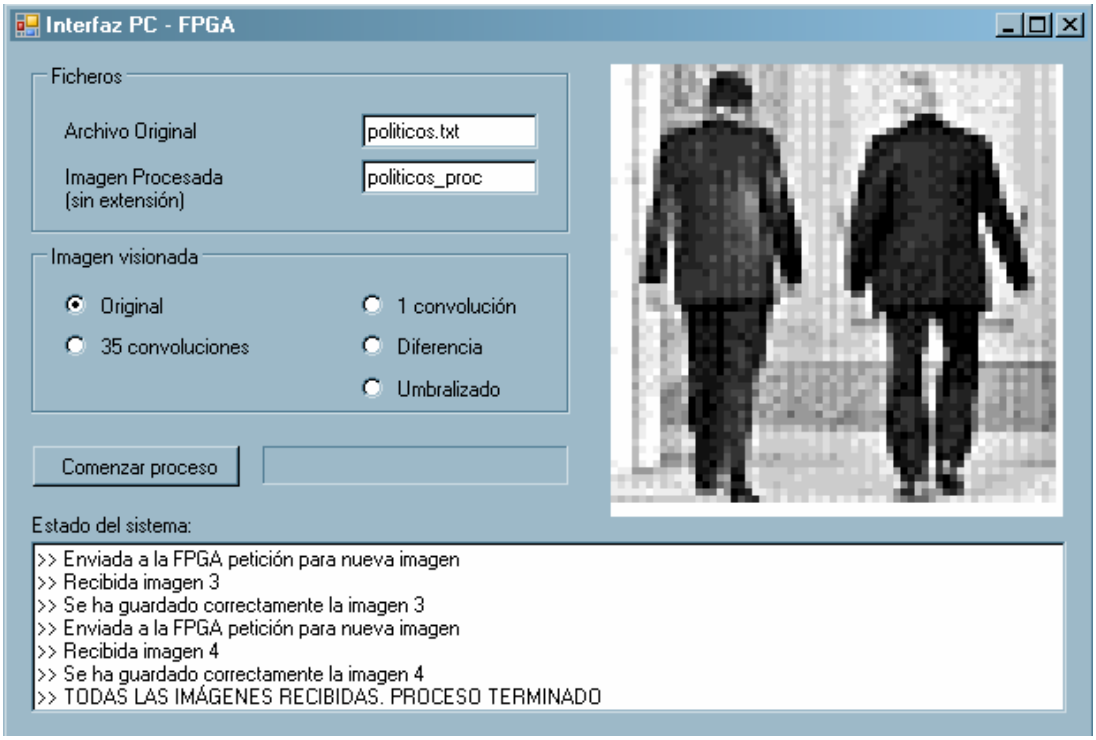


Figura 5.8. Imagen ruidosa enviada desde el PC a la FPGA.

Las imágenes resultantes del procesado:

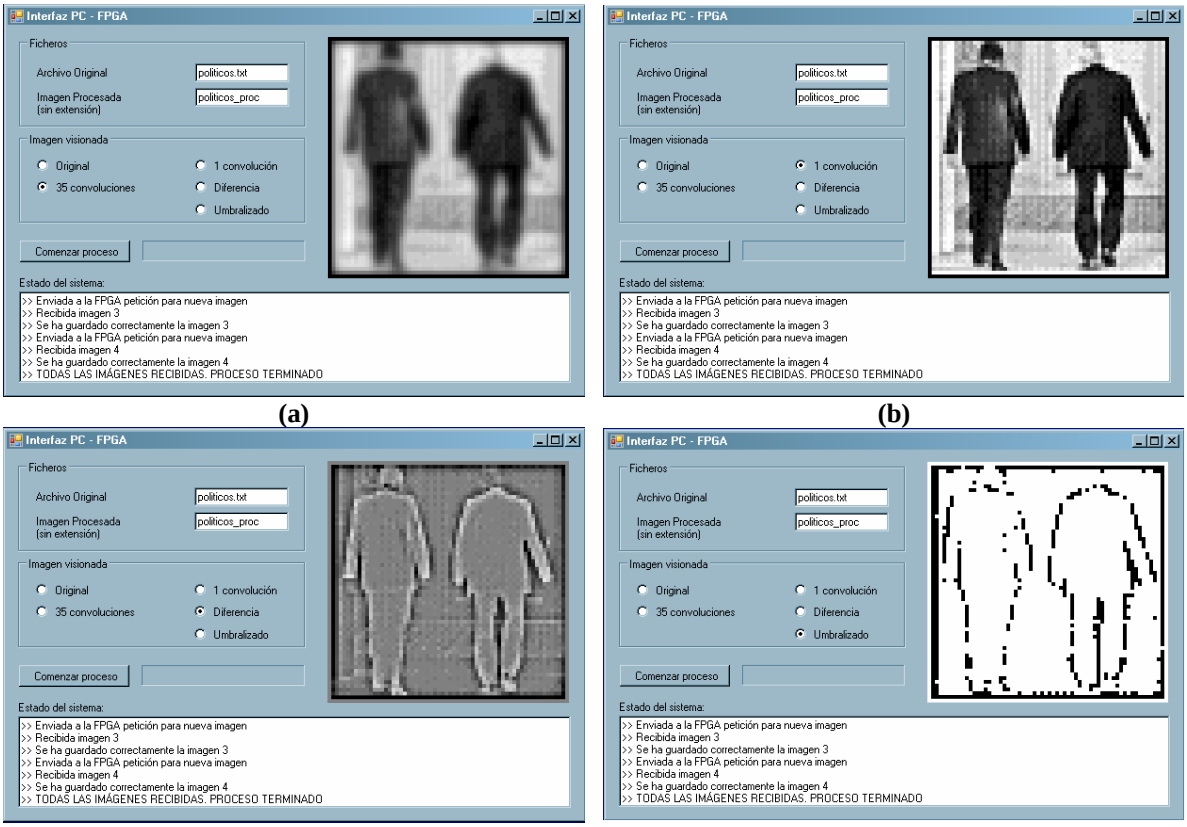


Figura 5.9. Imágenes resultantes del procesamiento por parte del sistema de visión.

La figura anterior muestra las imágenes enviadas desde la FPGA al PC, como resultado de procesar la imagen de la figura 5.8:

(a) Imagen resultante de aplicar 35 convoluciones a la original**.**
(b) Imagen resultante de aplicar 1 convolución a la original**.**
(c) Imagen diferencia de las dos anteriores**.**
(d) Imagen umbralizada**.**

Según los resultados obtenidos podemos concluir que este ejemplo de procesado es sensible al ruido presente en la imagen original, como puede observarse en las imágenes umbralizadas, en las que los bordes tienen una mayor definición para imágenes con un mayor grado de nitidez.

6. DISCUSIÓN Y CONCLUSIONES

Llegados a este punto, hemos implementado sobre una plataforma reconfigurable, un sistema de visión en un único chip con hardware específico para el procesamiento de imágenes. Hemos realizado una serie de pruebas sobre el sistema para verificar su funcionalidad. El siguiente paso es la caracterización de sus prestaciones y limitaciones. A partir de estos datos podremos hacer predicciones para ayudar a diseñar sistemas de visión on-chip en un futuro.

6.1 Caracterización del sistema de visión: prestaciones y limitaciones

Este sistema prototipo no tiene, en principio, limitaciones sobre el tamaño de la imagen. Es decir, existe un tamaño de imagen prefijado, con el cual se ha configurado el hardware, pero se puede modificar a conveniencia. La única limitación aparecería en el compromiso entre el tamaño de la imagen y las especificaciones temporales de la aplicación en tiempo real. De modo que trataremos de evaluar las prestaciones del sistema desde el punto de vista de su capacidad de cómputo. Si esta capacidad se aplica finalmente a un número reducido de imágenes más grandes, o a un conjunto masivo de imágenes más pequeñas, no va a ser objeto de nuestra discusión.

Tras los primeros tests sobre el sistema de visión, hemos calculado el número de ciclos de reloj empleados por cada uno de los operadores integrados en el coprocesador visual. Expresado en el número de ciclos por píxel, con el fin de extender los resultados a imágenes de cualquier tamaño, tenemos:

Operación	*Ciclos / píxel*
Convolución	14
Suma / Resta	4
Umbralizado	3

Tabla 6.1. Potencia de cálculo del coprocesador

Con el fin de evaluar las prestaciones y limitaciones de nuestro sistema, vamos a suponer que estuviera conectado a una cámara, CMOS o CCD, que actúa como fuente de imágenes. Consideremos un flujo de imágenes en tamaño QCIF (176x144 píxeles) con un *frame rate* de 25fps, es decir, la cámara envía 25 imágenes cada segundo. Vamos a aplicar los datos de la tabla 6.1 a la realización de 36 operaciones de convolución con máscara espacial, de 3x3 píxeles, una sustracción y un umbralizado sobre la imagen original. Esto quiere decir que necesitaríamos 511 ciclos de reloj por cada píxel. O sea, 12.63Mciclos de reloj por cada imagen. Si trabajamos con un reloj de 50MHz, que tiene un periodo de 20ns, necesitaríamos 253ms para procesar cada imagen, o lo que es lo mismo, un *frame rate* de 3.96fps. Esto aparece reflejado en la primera fila de la tabla 6.2. Como vemos, el flujo de imágenes resultado está por debajo del flujo de imágenes entrantes. Esto quiere decir que este hardware resulta insuficiente para esta aplicación[13], por lo que necesitaríamos bien un hardware que fuera capaz de realizar estas operaciones más rápido, o bien multiplexar

[13] Obsérvese que si el tamaño de la imagen fuera de 64x64, el *frame rate* alcanzado sería de 25.93 fps, permitiendo al sistema de visión realizar este tipo de aplicaciones

espacialmente, o sea, replicar, el hardware de procesamiento de que disponemos con el fin de paralelizar en parte el esfuerzo computacional.

6.2 Propuesta para el desarrollo de aplicaciones de visión

Va a ser muy habitual que nos enfrentemos a aplicaciones de visión cuyas tareas de procesamiento requieran una mayor capacidad de cómputo. En otras palabras, el *frame rate* requerido para un determinado tamaño de la imagen va a estar por encima de lo que puede ofrecernos el haber incluido un único convolucionador en nuestro sistema. Hablamos del convolucionador porque, como puede verse en la tabla 6.1, es el bloque que realiza un mayor esfuerzo computacional. Para que nuestro sistema alcance velocidades de procesamiento superiores, y por tanto, pueda enfrentarse a aplicaciones que con imágenes relativamente grandes tengan un *frame rate* por encima de los 3 ó 4 fps, nos planteamos la multiplexación espacial del principal módulo de procesamiento del coprocesador, es decir, el convolucionador. En concreto, se trata de que el convolucionador contenga más de un elemento de procesamiento permitiendo la operación sobre varios píxeles al mismo tiempo. La otra opción hubiera sido contemplar el uso de un operador de convolución que realizara esta misma operación sobre cada uno de los píxeles pero en menor tiempo. Sin embargo, para la plataforma reconfigurable en la que hemos diseñado el sistema de visión, hemos tratado de optimizar la capacidad de cálculo del hardware disponible. Es posible que algunas consideraciones de carácter arquitectural pudieran acelerar algo la operación del elemento de convolución, pero, dentro de lo que conocemos, el elemento que hemos diseñado está apurando las capacidades de la tecnología en la que está fabricada la FPGA. Por tanto, lo que vamos a hacer es replicar el hardware del elemento de convolución para que el bloque convolucionador pueda procesar varios píxeles en paralelo. Esto supondría una mejora en los tiempos de procesado, pero también un aumento en el consumo de recursos de la FPGA. La tabla 6.2 muestra una estimación de la mejora en las prestaciones del sistema y su efecto en el uso de los recursos de la FPGA. Para realizar la estimación de los recursos ocupados por los bloques a replicar, es decir, los correspondientes al convolucionador, hemos ejecutado la síntesis del sistema completo sin el convolucionador. De esta forma, obtenemos los recursos ocupados por el hardware a replicar comparando los datos de la síntesis sin el convolucionador con los que obtuvimos de la síntesis del sistema completo. La diferencia de los valores adquiridos en ambos casos va a ayudarnos a estimar la cantidad de recursos ocupados por el módulo a replicar. No vamos a considerar en este punto la posible problemática asociada con el acceso a una memoria de imágenes por parte de un conjunto de elementos de convolución de manera simultánea, ocupándonos en exclusiva de la estimación de la potencia de cómputo asociada al uso de un número variable de elementos de convolución. Para realizar esta valoración vamos a suponer que el procesamiento en paralelo de los diferentes elementos de convolución acelera al sistema de visión, en lo relativo a la velocidad de procesamiento de imágenes del mismo, de manera proporcional al número de elementos de convolución que consideremos en cada caso, sin tener en cuenta el efecto de posibles irregularidades en la colocación de los mismos sobre la imagen a tratar. Estos efectos podrían aparecer, por ejemplo, si trabajamos con un número de elementos que no puedan distribuirse de manera uniforme según las dimensiones de la imagen a procesar. Al margen de estas consideraciones, los cálculos realizados, mostrados en la tabla 6.2,

nos conducen a un caso límite práctico, impuesto por las características de la FPGA usada, en el que el sistema estaría equipado con 23 elementos de convolución, que, trabajando en paralelo, sería capaces de alcanzar una potencia de cómputo tal que permitiera al sistema un flujo de imágenes procesadas de 91.05fps.

Nº conv.	Slices (% del total)	Flip flop (%)	LUT (%)	Mult. (%)	fps
1	4133 (30%)	2144 (6%)	7632 (27%)	3 (2%)	3.96
4	5426 (39%)	2855 (10%)	9879 (36%)	6 (4%)	15.8
9	7581 (55%)	4040 (15%)	13624 (50%)	11 (8%)	35.63
16	10598 (77%)	5699 (21%)	18867 (69%)	18 (13%)	63.34
23	13615 (~100%)	7358 (27%)	24110 (88%)	25 (18%)	91.05

Tabla 6.2 Estimación de recursos consumidos y del *frame rate* alcanzado por el sistema

En la primera columna aparece el número de píxeles sobre los que actúa en paralelo el bloque convolucionador. En la segunda columna tenemos el número de *slices* de la FPGA que se han utilizado. Los *slices* son las unidades lógicas reconfigurables básicas en las que se agrupan las puertas dentro de la FPGA. Finalmente el número total de *slices* disponibles va a ser el factor limitante del número de elementos de convolución que va a ser posible incluir en nuestro sistema. A continuación, las tres columnas siguientes muestran el uso de flip-flops, LUT's y multiplicadores que se hace. Aparecen aquí porque también los estuvimos considerando por si finalmente resultaban ser los factores limitantes. Y por último, a la derecha de la tabla, se muestra la estimación del número de imágenes por segundo que el sistema va a ser capaz de manejar. Para el diseño de sistemas de visión ajustados a las especificaciones de una aplicación concreta, esta tabla va a sernos de mucha utilidad.

6.3 Conclusiones

Se han estudiado los fundamentos del procesamiento digital de imágenes y los operadores y funcionalidades básicas necesarias para el desarrollo de aplicaciones de visión artificial.

Se ha diseñado un sistema de visión en un único chip basado en una plataforma reconfigurable y con hardware específico para acelerar la ejecución de tareas de procesamiento de imágenes.

Se ha utilizado para ello una arquitectura compatible con bloques de propiedad intelectual en silicio de uso extendido, con el fin de mostrar la modularidad del sistema y las posibilidades de ampliación.

Se han diseñado los elementos necesarios, no estándar, para el procesamiento de imágenes, se han incluido en el sistema mediante la interfaz adecuada, y se ha comprobado su funcionamiento y su compatibilidad con el resto de los elementos del sistema.

Se ha implementado el sistema diseñado en una FPGA, en concreto una Virtex II Pro, utilizando una placa de desarrollo, y se ha verificado su funcionamiento.

Se ha desarrollado una aplicación, controlada desde el PC, para poner de manifiesto las capacidades de procesamiento y configurabilidad del sistema.

Se han caracterizado las prestaciones y limitaciones del sistema de visión y se han hecho predicciones basadas en esta evaluación que serán de utilidad en el diseño de sistemas ajustados a unas especificaciones reales.

7. BIBLIOGRAFÍA

[1] *Rec. H. 261: Video Codec for Audiovisual Services at p x 64kbit/s*. Rec. of the ITU, Helsinki, Finland 1993.

[2] *Intel® Pentium® 4 Processor on 0.13 Micron Process Datasheet*. Doc. No.: 298643-012, Feb. 2004.

[3] *AMD Athlon™ 64 FX Product Data Sheet*. Document No. 30431, May 2004 (http://www.amd.com/us-n/assets/content_type/white_papers_and_tech_docs/30431.pdf).

[4] *SH7705 Group Hardware Manual: Renesas 32-Bit RISC Microcomputer SuperH_ RISC engine Family/ SH7700 Series*. Revision 2.00. Sept. 2003 (http://documentation.renesas.com/eng/products/mpumcu/rej09b0082_sh7705.pdf).

[5] R. E. Amtower, "Low Cost Machine Vision Systems Extend Manufacturing Applications" *WESCON/'93. Conference Record*, pp. 116- 119, 28-30 Sept. 1993.

[6] M. Sugisaka, Wang Xin, "A Complex Control Method for an Intelligent Mobile Vehicle". Proceedings of the 1996 IEEE Intelligent Vehicles Symposium, pp. 53-57, 19-20 Sept. 1996.

[7] G. P. Stein et al., "A Computer Vision System on a Chip: a Case Study from the Automotive Domain". *Proc. 2005 IEEE Comp. Soc. Conf. on Comp. Vision and Pattern Recognition, Vol. 3*, pp. 130-134, June 2005.

[8] T. Makimoto, T. T. Doi, "Chip Technologies for Entertainment Robots - Present and Future". *Int. Electron Devices Meeting*, pp. 9-16, Dec. 2002.

[9] Legal-Ayala, H.A., Facon, J. "Automatic segmentation of brain MRI through learning by example" *Image Processing, 2004. ICIP apos; 04. 2004 International Conference on Volume 2*, pp. 917-920, 24-27 Oct. 2004

[10] K. Park, D. Metaxas and L. Axel. "Cardiac Motion Analysis from Tagged MR Images using a Generic model", *BMES/EMBS*, Oct. 2004.

[11] J. Zhang, C. I. Chang, Miller, S.J., Kang, K.A. "Optical biopsy of skin tumors" *21st Annual Conf. and the 1999 Annual Fall Meeting of the Biomedical Engineering Soc. BMES/EMBS Conference, 1999. Proceedings of the First Joint Volume 2*, pp.1095, 13-16 Oct. 1999

[12] J. Dengler, S. Behrens, J. F. Desaga. "Segmentation of microcalcifications in mammograms" *Medical Imaging, IEEE Transactions on Volume 12, Issue 4*, pp. 634-642. Dec. 1993.

[13] L. Vincent and B. Masters (1992). "Morphological image processing and network analysis of cornea endothelial cell images". *In Image algebra and morphological image processing III*, P. Gader, E. Dougherty, and J. Serra, eds., Vol. SPIE-1769, pp. 212-226.

[14] J.-P. Thiran, B. Macq, J. Mairesse. "Morphological classification of cancerous cells" *Image Processing, 1994. Proceedings. ICIP-94., IEEE International Conference Volume 3*, pp. 706 – 710, 13-16 Nov. 1994.

[15] M. Pesaresi, J. A. Benediktsson. "A new approach for the morphological segmentation of high-resolution satellite imagery". *Geoscience and Remote Sensing, IEEE Transactions on Volume 39, Issue 2*, pp. 309 – 320. Feb 2001.

[16] R. B. Starkey, I. Aleksander. "Facial recognition for police purposes using computer graphics and neural networks" *Electronic Images and Image Processing in Security and Forensic Science, IEE Colloquium*, pp. 2/1 - 2/2. 22 May 1990

[17] B. Rosen and L. Vincent. "Morphological Image Processing Techniques Applied to Detection of Correlogram Tracks". *U.S. Navy Journal of Underwater Acoustics*, 1994.

[18] Y. Liu, P. Payeur. “Vision-based detection of activity for traffic control”. *Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference on Volume 2*, pp. 1347 – 1350. 4-7 May 2003
[19] D. Bloomberg and L. Vincent, (1995). “Blur hit-miss transform and its use in document image pattern detection”. In Document Recognition II, L. Vincent and H. Baird, eds., Vol. SPIE-2422, pp. 278.292.
[20] B. Modayur, V. Ramesh, R. Haralick, and L. Shapiro. “MUSER: a prototype musical score recognition system using mathematical morphology”. *Machine Vision and Applications*, 1993.
[21] E. Aarts, R. Roovers, "IC Design Challenges for Ambient Intelligence", *Design, Aut. and Test in Europe*, pp. 2-7, March 2003.
[22] R. C. González and P. Wintz, “*Digital Image Processing*”, Addison-Wesley, Reading MA, 1987.
[23] L.O. Chua and L. Yang, “Cellular Neural Networks: Applications”, *IEEE Transactions on Circuits and Systems-I: Fundamental Theory and Applications, Vol. 35, No. 10*, pp. 1273-1290, October 1988.
[24] T. Aitch, *A Pipelined RISC CPU: Aquarius. Rev. 1.1*, July 2003.
[25] Renesas Technology. *SH-1/SH-2/SH-DSP Software Manual, Rev. 5.00*, June 30, 2004.
[26] Wade D. Peterson, *Spec. for the WISHBONE SoC Interconnection Architecture for Portable IP Cores. Rev. B. 3.* Sept. 2002.
[27] “*Using Block RAM in Spartan 3 Generation FPGAs*”, (**www.xilinx.com**)
[28] *Digilent Inc. Xilinx University Program Virtex-II Pro Development System - Hardware Reference, Ver. 1.0*, March 2005.
[29] B. Jähne, H. Haußecker, P. Geißler (Eds.), *Handbook of Computer Vision and Applications. Vol. 2*. Academic Press, San Diego, 1999.

8. APÉNDICES

8.1 Códigos Verilog

Esquema de archivos

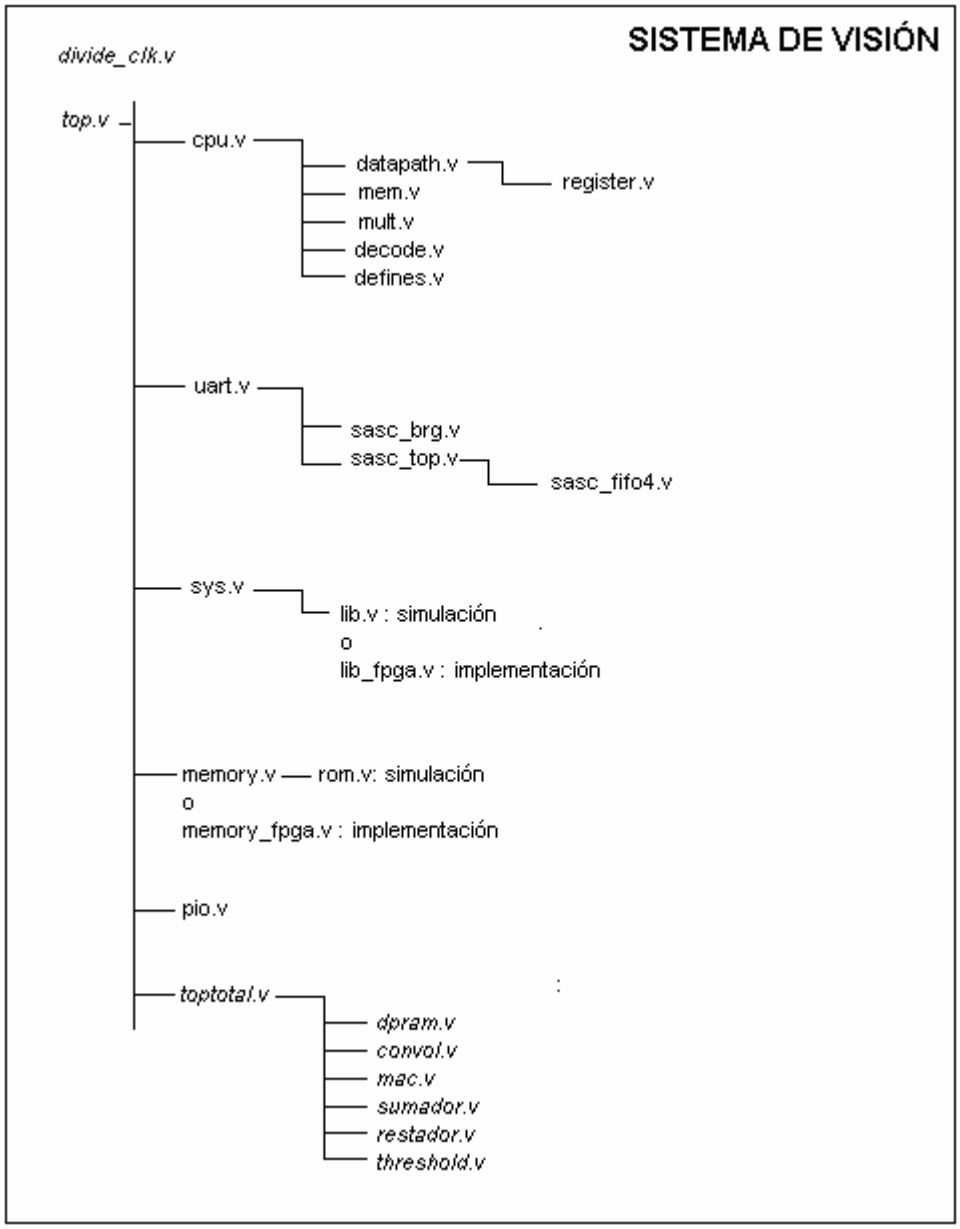


Figura 8.1. Esquema de ficheros Verilog del sistema.

NOTA: En el esquema anterior, se muestran en cursiva los archivos cuyos códigos han sido modificados o añadidos a la estructura de Aquarius, que se detallan a continuación. El resto pueden consultarse en OpenCores.org [24]

Archivos originales del coprocesador visual o modificados de Aquarius

Top.v : Nivel jerárquico superior de Aquarius y del sistema de visión.

Divide_clk.v: Bloque encargado de dividir la frecuencia de reloj.

Toptotal.v : Nivel más alto del coprocesador visual.

Convol.v : Bloque *direccionador* del convolucionador.

Sumador.v : Sumador de imágenes.

Restador.v : Restador de imágenes.

Mac.v: Bloque *actualizador del nivel del píxel* del convolucionador.

Threshold.v : Umbralizador.

Top.v

```
//=====================================================
// Aquarius Project
//    SuperH-2 ISA Compatible RISC CPU
//-----------------------------------------------------
// Module      : Top Layer (A Simple MCU)
//-----------------------------------------------------
// File        : top.v
// Library     : none
// Description : Top Layer for Aquarius.
//               It is a simple MCU Chip.
// Simulator   : Icarus Verilog (Cygwin)
// Synthesizer : Xilinx XST (Windows XP)
// Author      : Thorn Aitch
//-----------------------------------------------------
// Revision Number : 1
// Date of Change  : 15th April 2002
// Creator         : Thorn Aitch
// Description     : Initial Design
//-----------------------------------------------------
// Revision Number : 2
// Date of Change  : 30th April 2003
// Modifier        : Thorn Aitch
// Description     : Release Version 1.0
//=====================================================
// Copyright (C) 2002-2003, Thorn Aitch
//
// Designs can be altered while keeping list of
// modifications "the same as in GNU" No money can
// be earned by selling the designs themselves, but
// anyone can get money by selling the implementation
// of the design, such as ICs based on some cores,
// boards based on some schematics or Layouts, and
// even GUI interfaces to text mode drivers.
// "The same as GPL SW" Any update to the design
// should be documented and returned to the design.
// Any derivative work based on the IP should be free
// under OpenIP License. Derivative work means any
// update, change or improvement on the design.
// Any work based on the design can be either made
// free under OpenIP license or protected by any other
// license. Work based on the design means any work uses
// the OpenIP Licensed core as a building black without
// changing anything on it with any other blocks to
// produce larger design.  There is NO WARRANTY on the
// functionality or performance of the design on the
// real hardware implementation.
// On the other hand, the SuperH-2 ISA (Instruction Set
// Architecture) executed by Aquarius is rigidly
// the property of Renesas Corp. Then you have all
// responsibility to judge if there are not any
// infringements to Renesas's rights regarding your
// Aquarius adoption into your design.
// By adopting Aquarius, the user assumes all
// responsibility for its use.
// This project may cause any damages around you, for
// example, loss of properties, data, money, profits,
// life, or business etc. By adopting this source,
// the user assumes all responsibility for its use.
//=====================================================

`include "timescale.v"
`include "defines.v"

//******************************
// Top Module
//******************************
module top(
    CLK_SRC, RST_n,
    LCDRS, LCDRW, LCDE,
           LCDDBO, LCDDBI,
           KEYYO, KEYXI,
    RXD, TXD//, CTS, RTS
   );
```

```
    input  CLK_SRC; // non stop clock
    input  RST_n;
    output LCDRS;
    output LCDRW;
    output LCDE;
    output [7:0] LCDDBO;
    input  [7:0] LCDDBI;
    output [4:0] KEYYO;
    input  [4:0] KEYXI;
    input  RXD;
    output TXD;
//    input  CTS;
//    output RTS;



//------------------
// Internal Signals
//------------------
    wire   CLK;          // internal system clock, which stops during sleep
    reg    RST;          // 2nd sync reset
    reg    RST1;         // 1st sync reset
    wire   CYC;          // external bus cycle to be kept
    wire   STB;          // external bus strobe
    reg    ACK;          // external device acknowledge
    wire   [31:0] ADR;   // external address
    reg    [31:0] DATI;  // external data read bus
    wire   [31:0] DATO;  // external data write bus
    wire   WE;           // external write/read
    wire   [3:0] SEL;    // external data valid position
    reg    IF_WIDTH;     // IF_WIDTH : external fetch space width

    reg    CEMEM, CEPIO, CEUART, CESYS, CECONV, CESUM, CERES, CETHR, CEDPRAM;  //chip enable of each
                                                                                device
    wire   [31:0] DATMEM;  // direct memory output
    wire   [31:0] DATPIO;  // direct pio output
    wire   [31:0] DATUART; // direct uart output
    wire   [31:0] DATSYS; // direct sys output
    wire   [31:0] DATCOP; // direct coprocessor DPRAM output
    wire   [31:0] PI;  // port input
    wire   [31:0] PO;  // port output

    wire   RXD, TXD, CTS, RTS; //uart signals

    wire   [2:0] EVENT_REQ;   // Hardware Exception Event Request
    wire   EVENT_ACK;         // Hardware Exception Event Acknowledge
    wire   [11:0] EVENT_INFO; // Hardware Exception Event Information

    wire   SLP;       // SLEEP output

    wire  CLK_20M;


    assign LCDRS     = PO[8];
    assign LCDRW     = PO[9];
    assign LCDE      = PO[10];
    assign LCDDBO    = PO[7:0];
    assign PI[ 7: 0] = LCDDBI;
    assign KEYYO     = PO[20:16];
    assign PI[20:16] = KEYXI;
    assign PI[31:21] = 11'b00000000000;
    assign PI[15: 8] =  8'b00000000;

    assign CTS = RTS;

//********
// Modules
//********
    cpu CPU(
        // system signal
        .CLK(CLK), .RST(RST),
        // WISHBONE external bus signal
        .CYC_O(CYC), .STB_O(STB), .ACK_I(ACK),
        .ADR_O(ADR), .DAT_I(DATI), .DAT_O(DATO),
        .WE_O(WE), .SEL_O(SEL),
```

```
        .TAG0_I(IF_WIDTH),
        // Exception
        .EVENT_REQ_I(EVENT_REQ),
        .EVENT_ACK_O(EVENT_ACK),
        .EVENT_INFO_I(EVENT_INFO),
        //SLEEP
        .SLP_O(SLP)
    );

    memory MEMORY(
        .CLK(CLK), .CE(CEMEM), .WE(WE), .SEL(SEL),
        .ADR(ADR[13:0]), .DATI(DATO), .DATO(DATMEM)
    );

    pio PIO(
        .CLK(CLK), .RST(RST),
        .CE(CEPIO), .WE(WE), .SEL(SEL),
        .DATI(DATO), .DATO(DATPIO),
        .PI(PI), .PO(PO)
    );

    uart UART(
        .CLK(CLK), .RST(RST),
        .CE(CEUART), .WE(WE), .SEL(SEL),
        .DATI(DATO), .DATO(DATUART),
        .RXD(RXD), .TXD(TXD), .CTS(CTS), .RTS(RTS)
    );

      divide_clk DIVIDE_CLK(
         .CLK(CLK_SRC), .RST_n(RST_n),
         .CLK_int(CLK_20M)
              );

    sys SYS(
        .CLK_SRC(CLK_20M), .CLK(CLK), .SLP(SLP), .WAKEUP(~KEYXI[4]), .RST(RST),
        .CE(CESYS), .WE(WE), .SEL(SEL), .ACK(ACK),
        .DATI(DATO), .DATO(DATSYS),
        .EVENT_REQ(EVENT_REQ),
        .EVENT_ACK(EVENT_ACK),
        .EVENT_INFO(EVENT_INFO),
        .STB(STB), .ADR(ADR)
    );

       toptotal COP(
        .clk(CLK),  .rst(RST), .address(ADR[15:0]),
        .dati(DATO), .cesum(CESUM), .ceres(CERES),
        .cethr(CETHR), .ceconv(CECONV), .cs_0(CEDPRAM),
        .we_0(WE),  .sel(SEL),  .dato(DATCOP)
    );

//************
// Address MAP
//************
// address          size wait width device
// 00000000-0000FFFF 64K  0    32    MEMORY (shadow every 16KB)
// 00010000-0001FFFF 64K  3    32    MEMORY (shadow every 16KB)
// 00020000-0002FFFF 64K  0    16    MEMORY (shadow every 16KB)
// 00030000-0003FFFF 64K  3    16    MEMORY (shadow every 16KB)
// 00040000-ABCCFFFF .................(shadow MEMORY)
// ABCD0000-ABCD00FF 256  3    32    PIO (shadow every 4B)
// ABCD0100-ABCD01FF 256  3    32    UART(shadow every 4B)
// ABCD0200-ABCD02FF 256  3    32    SYS (shadow every 8B)
// ABCD0300-ABCD03FF 256  3    32    CONVOL
// ABCD0400-ABCD04FF 256  3    32    SUM
// ABCD0500-ABCD05FF 256  3    32    THR
// ABCD0600-ABCD06FF 256  3    32    RES
// ABCD0700-FFFBFFFF .................(shadow MEMORY)
// FFFC0000-FFFCFFFF 64K  0    32    MEMORY (shadow every 16KB)
// FFFD0000-FFFDFFFF 64K  3    32    MEMORY (shadow every 16KB)
// FFFE0000-FFFEFFFF 64K  0    16    MEMORY (shadow every 16KB)
// FFFF0000-FFFFFFFF 64K  3    16    MEMORY (shadow every 16KB)
//
// <MEMORY>
// ****0000-****1FFF  8K  ROM
// ****2000-****3FFF  8K  RAM
// ****4000-****5FFF  8K  ROM (shadow)
```

```
// ****6000-****7FFF  8K  RAM (shadow)
// ****8000-****9FFF  8K  ROM (shadow)
// ****A000-****BFFF  8K  RAM (shadow)
// ****C000-****DFFF  8K  ROM (shadow)
// ****E000-****FFFF  8K  RAM (shadow)

    always @(posedge CLK)
    begin
        RST1 <= ~RST_n;
        RST  <= RST1;
    end

    always @(DATMEM or DATPIO or DATUART or DATSYS or DATCOP) begin
      DATI <= DATMEM | DATPIO | DATUART | DATSYS | DATCOP; // read data gathering
    end

    always @(STB or ADR)
    begin
        if (STB == 1'b0)
            {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b000000000;
        else if (ADR[31:8] == 24'hABCD00)
            {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b010000000;
        else if (ADR[31:8] == 24'hABCD01)
            {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b001000000;
        else if (ADR[31:8] == 24'hABCD02)
            {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b000100000;
        else if (ADR[31:8] == 24'hABCD03)
            {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b000010000;
        else if (ADR[31:8] == 24'hABCD04)
            {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b000001000;
        else if (ADR[31:8] == 24'hABCD05)
            {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b000000100;
        else if (ADR[31:8] == 24'hABCD06)
            {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b000000010;
         else if (ADR[31:16] == 16'hFFFD)
            {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b000000001;
         else
            {CEMEM,CEPIO,CEUART,CESYS,CECONV,CESUM,CETHR,CERES,CEDPRAM} <= 9'b100000000;
    end

//---------------------
// Control Access Cycle
//---------------------
    reg ACK0; // 0 wait device
    reg ACK3; // 3 wait device

    always @(ACK0 or ACK3) begin
        ACK <= ACK0 | ACK3;
    end

    always @(STB or ADR) begin
        if ((STB == 1'b1) && (ADR[16] == 1'b0))
            ACK0 <= 1'b1;
        else
            ACK0 <= 1'b0;
    end

    // S0: ACK3=0
    // S1: ACK3=0
    // S2: ACK3=0
    // S3: ACK3=1
    //
    // S0 -> S0
    // S0 -> S1 if ADR = **8 ~ **F, then wait
    // S2 -> S3
    // S3 -> S0

    reg [1:0] ACK3_STATE;
    reg [1:0] ACK3_NEXT_STATE;
    always @(posedge CLK or posedge RST) begin
        if (RST == 1'b1)
            begin
                ACK3_STATE <= 2'b00;
            end
        else
            begin
```

```
            ACK3_STATE <= ACK3_NEXT_STATE;
        end
    end
    always @(ACK3_STATE or STB or ADR) begin
        case (ACK3_STATE)
          2'b00 : if ((STB & ADR[16])  == 1'b1)
                    ACK3_NEXT_STATE <= 2'b01;
                  else
                    ACK3_NEXT_STATE <= 2'b00;
          2'b01 : ACK3_NEXT_STATE <= 2'b10;
          2'b10 : ACK3_NEXT_STATE <= 2'b11;
          2'b11 : ACK3_NEXT_STATE <= 2'b00;
          default : ACK3_NEXT_STATE <= 2'bxx;
        endcase
    end
    always @(ACK3_STATE)
        ACK3 <= ACK3_STATE[1] & ACK3_STATE[0];

//-------------------
// Control Data Width
//-------------------
    always @(ADR) begin
        if (ADR[17] == 1'b0)
            IF_WIDTH <= 1'b1;
        else
            IF_WIDTH <= 1'b0;
    end

//=====================================================
  endmodule
//=====================================================
```

Divide_clk.v

```
//////////////////////////////////////////////////////////////////////////////////////////////////////
// Engineer:   Jose Fernandez Perez
//
// Create Date:    19:44:19 03/28/2006
// Module Name:  divide_clk.v
// Project Name:  Aquarius
// Target Devices: X2CVP30
// Tool versions: Xilinx ISE v8.1
// Description: This function divides by 2 the target's clock fraquency
// Revision 0.01 - File Created
//
//////////////////////////////////////////////////////////////////////////////////////////////////////
module divide_clk(CLK,RST_n,CLK_int);
        input CLK;
        input RST_n;
        output CLK_int;

        wire CLK;
        wire RST;

        reg CLK_int;

        always @ (posedge CLK or negedge RST_n)
        begin
                if (RST_n == 0)
                        CLK_int <= 1;
                else
                        CLK_int <=~CLK_int;
        end

endmodule
```

Toptotal.v

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Engineer:  Jose Fernandez Perez
//
// Create Date:    19:44:19 03/28/2006
// Module Name:    toptotal.v
// Project Name:   Aquarius
// Target Devices:  X2CVP30
// Tool versions:  Xilinx ISE v8.1
// Description:   Top layer of the visual coprocessor
// Revision 0.01 - File Created
//
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module toptotal(clk,rst,address,dati,cesum,ceres,cethr,ceconv,cs_0,we_0, sel,dato);
    input clk;
    input rst;
    input [15:0] address;
    input [31:0] dati;          //wishbone signal: Aquarius to coprocessor data
    input cesum;
    input ceres;
    input cethr;
    input ceconv;
    input cs_0;
    input we_0;
    input [3:0] sel;            //wishbone signal: data valid position
    output [31:0] dato;         //wishbone signal: coprocessor to Aquarius data


                wire [15:0] datin;
                wire [7:0] datmem;
                reg cs_1;
                reg we_1;
                wire cs_1t;
                wire we_1t;
                wire cs_1s;
                wire cs_1r;
                wire we_1s;
                wire we_1r;
                wire cs_1c;
                wire we_1c;
                reg [31:0] dato;

                reg [7:0] data_1_in;
                wire [7:0] data_1_int;
                wire [7:0] data_1_ins;
                wire [7:0] data_1_inr;
                wire [7:0] data_1;
                wire [2:0] addressinthr;
                wire [7:0] datain;
                reg [15:0] addr1;
                wire [15:0] addr1t;
                wire [15:0] addr1s;
                wire [15:0] addr1r;
                wire [15:0] addr1c;
                wire [3:0] addressinsum;
                wire [3:0] addressinres;
                wire [3:0] addressinconv;
                wire [7:0] data_1_inc;
                wire [16:0] resultado;
                wire clk_en;
                wire aclr;
                wire res_listo;
                wire [7:0] db;

                assign addressinconv[3:0] = address[3:0];
                assign addressinsum[3:0] = address[3:0];
                assign addressinres[3:0] = address[3:0];
                assign datain[7:0] = datin[7:0];
                assign addressinthr[2:0] = address[2:0];
                assign data_1_inc[7:0] = resultado[7:0];
                assign datin[15:0] = dati[15:0];

//modules
```

```
    convol CONVOL(clk,rst,res_listo,datin,addressinconv,ceconv,we_0,addr1c,we_1c,cs_1c,aclr,db,clk_en);
    mac MAC(data_1,db,clk,rst,aclr,clk_en,resultado,res_listo);
    sumador SUMADOR(clk,rst,cesum,we_0,data_1,datin,addressinsum,we_1s,cs_1s,addr1s,data_1_ins);
    restador RESTADOR(clk,rst,ceres,we_0,data_1,datin,addressinres,we_1r,cs_1r,addr1r,data_1_inr);
    threshold THRESHOLD(datin,addressinthr,clk,rst,data_1,cethr,we_0,data_1_int,addr1t,we_1t,cs_1t);
    dpram DPRAM(address,addr1,clk,clk,datain,data_1_in,datmem,data_1,cs_0,cs_1,we_0,we_1);


 always @(cs_0 or datmem or sel or we_0)
   begin
      if ((cs_0 == 1'b1) && (we_0==1'b0) && (sel[3:0]==4'b1000))
        begin
            dato[31:24] <= datmem;
            dato[23:16] <= 8'h00;
            dato[15:8] <= 8'h00;
            dato[7:0] <= 8'h00;
         end
       else if ((cs_0 == 1'b1) && (we_0==1'b0) && (sel[3:0]==4'b0100))
        begin
            dato[31:24] <= 8'h00;
            dato[23:16] <= datmem;
            dato[15:8] <= 8'h00;
            dato[7:0] <= 8'h00;
         end
       else if ((cs_0 == 1'b1) && (we_0==1'b0) && (sel[3:0]==4'b0010))
        begin
            dato[31:24] <= 8'h00;
            dato[23:16] <= 8'h00;
            dato[15:8] <= datmem;
            dato[7:0] <= 8'h00;
          end
       else if ((cs_0 == 1'b1) && (we_0==1'b0) && (sel[3:0]==4'b0001))
        begin
           dato[31:24] <= 8'h00;
           dato[23:16] <= 8'h00;
           dato[15:8] <= 8'h00;
            dato[7:0] <= datmem;
        end
      else
        begin
           dato[31:0] <= 32'h00000000;
        end
   end

   always @(cs_1t or cs_1s or cs_1c or cs_1r) begin
     cs_1 <= cs_1t | cs_1s | cs_1c | cs_1r; // read data gathering
   end
   always @(we_1t or we_1s or we_1c or we_1r) begin
     we_1 <= we_1t | we_1s | we_1c | we_1r; // read data gathering
   end
   always @(addr1t or addr1s or addr1c or addr1r) begin
     addr1 <= addr1t | addr1s | addr1c | addr1r; // read data gathering
   end
    always @(data_1_int or data_1_ins or data_1_inc or data_1_inr) begin
     data_1_in <= data_1_int | data_1_ins | data_1_inc | data_1_inr; // read data gathering
   end
endmodule
```

Convol.v

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Engineer:   Jose Fernandez Perez
//
// Create Date:    19:44:19 03/28/2006
// Module Name:  convol.v
// Project Name:  Aquarius
// Target Devices: X2CVP30
// Tool versions: Xilinx ISE v8.1
// Description: This function performs a convolution operation in which a 3x3
//                    mask is passed to process an image
//
// Revision 0.01 - File Created
//
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module convol(clk,rst,res_listo,datin,address,ceconv,we,addr_o, we_1,ce_1,dato_leido,mascara,clk_en);
    input clk;
    input rst;
    input res_listo;                                                    //processed pixel
    input [15:0] datin;
    input [3:0] address;
    input ceconv;
    input we;                                                           //Aquarius wishbone signal
    output [15:0] addr_o;                                               //memory address
    output we_1;                                                        //memory write enable signal
    output ce_1;
    output dato_leido;
    output [7:0] mascara;
    output clk_en;


     reg [15:0] addr_o;
     reg we_1;
     reg ce_1;
     reg dato_leido;
     reg [7:0] mascara;
     reg clk_en;
     reg [7:0] DIV;

     reg [15:0] DIRECC_O;
     reg [15:0] p_DIRECC_O;
     reg [15:0] DIRECC_D;
     reg [15:0] p_DIRECC_D;
     reg [7:0] M;
     reg [7:0] N;
     reg [7:0] i;
     reg [7:0] p_i;
     reg [7:0] j;
     reg [7:0] p_j;
     reg inicioconv;
     reg p_inicioconv;
     reg [7:0] mask [0:8];

     reg [3:0] estado_act;
     reg [3:0] estado_sig;

 always @(posedge clk or posedge rst)
    begin
            if (rst==1'b1)
                    begin
                    estado_act<=2'b00;
                    M<=0;
                    N<=0;
                    DIRECC_O<=0;
                    DIRECC_D<=0;
                    i<=1;
                    j<=1;
                    inicioconv<=0;
                    mask[0]<=0;
                    mask[1]<=0;
                    mask[2]<=0;
                    mask[3]<=0;
                    mask[4]<=0;
                    mask[5]<=0;
```

```
                mask[6]<=0;
                mask[7]<=0;
                mask[8]<=0;
                end

        else
                begin
                estado_act<=estado_sig;
                i<=p_i;
                j<=p_j;
                DIRECC_O<=p_DIRECC_O;
                DIRECC_D<=p_DIRECC_D;
                inicioconv<=p_inicioconv;
                if(ceconv==1'b1 && we==1'b1)          //ceconv*we
                        begin
                          if (address==4'b1001) inicioconv<=datin[0];
                          else if (address==4'b1010) DIRECC_O<=datin;
                          else if (address==4'b1100) M<=datin[7:0];
                          else if (address==4'b1101) N<=datin[7:0];
                          else if (address==4'b1110) DIRECC_D<=datin;
                          else if (address==4'b0000) mask[0]<=datin[7:0];
                           else if (address==4'b0001) mask[1]<=datin[7:0];
                          else if (address==4'b0010) mask[2]<=datin[7:0];
                          else if (address==4'b0011) mask[3]<=datin[7:0];
                          else if (address==4'b0100) mask[4]<=datin[7:0];
                          else if (address==4'b0101) mask[5]<=datin[7:0];
                          else if (address==4'b0110) mask[6]<=datin[7:0];
                          else if (address==4'b0111) mask[7]<=datin[7:0];
                          else if (address==4'b1000) mask[8]<=datin[7:0];

                        end
                      end

      end


    always @(estado_act or mask[0] or mask[1] or mask[2]or mask[3] or mask[4] or mask[5] or mask[6] or mask[7] or
mask[8] or M or N or DIRECC_O or DIRECC_D or inicioconv or res_listo or i or j)
        begin
          case (estado_act)
             4'b0000:begin
                        we_1<=0;
                        addr_o<=0;
                        p_inicioconv<=inicioconv;
                       p_DIRECC_D<=DIRECC_D;
                        p_DIRECC_O<=DIRECC_O;
                        ce_1<=0;
                        p_i<=1;
                        p_j<=1;
                        dato_leido<=0;
                        mascara<=0;
                        clk_en<=0;
                        if (inicioconv==1'b1)
                           begin
                              p_DIRECC_D<=DIRECC_D+1+N;
                              p_DIRECC_O<=DIRECC_O;
                              we_1<=0;
                              ce_1<=1;
                              estado_sig=4'b0001;
                           end

                        else
                              estado_sig=4'b0000;
                       end

             4'b0001:begin
                        p_inicioconv<=0;
                        we_1<=0;
                        ce_1<=1;
                        p_i<=i;
                        p_j<=j;
                        p_DIRECC_D<=DIRECC_D;
                        p_DIRECC_O<=DIRECC_O;
                        addr_o<=DIRECC_O;
                        mascara<=0;
                        clk_en<=1;
```

```
            estado_sig=4'b0010;
            dato_leido<=0;
            end

4'b0010:begin
            p_inicioconv<=0;
            we_1<=0;
            ce_1<=1;
            p_i<=i;
            p_j<=j;
            clk_en<=1;
            p_DIRECC_D<=DIRECC_D;
            p_DIRECC_O<=DIRECC_O;
            dato_leido<=0;
            addr_o<=DIRECC_O+1;
            mascara<=mask[0];
            estado_sig=4'b0011;
            end

 4'b0011:begin
            p_inicioconv<=0;
            we_1<=0;
            ce_1<=1;
            p_i<=i;
            p_j<=j;
            clk_en<=1;
            dato_leido<=0;
            p_DIRECC_D<=DIRECC_D;
            p_DIRECC_O<=DIRECC_O;
            addr_o<=DIRECC_O+2;
            mascara<=mask[1];
            estado_sig=4'b0100;
           end

4'b0100:begin
            p_inicioconv<=0;
            we_1<=0;
            ce_1<=1;
            p_i<=i;
            p_j<=j;
            dato_leido<=0;
            p_DIRECC_D<=DIRECC_D;
            p_DIRECC_O<=DIRECC_O;
            clk_en<=1;
            addr_o<=DIRECC_O+N;
            mascara<=mask[2];
            estado_sig=4'b0101;
            end

4'b0101:begin
            p_inicioconv<=0;
            we_1<=0;
            ce_1<=1;
            p_i<=i;
            p_j<=j;
            p_DIRECC_D<=DIRECC_D;
            p_DIRECC_O<=DIRECC_O;
            dato_leido<=0;
            clk_en<=1;
            addr_o<=DIRECC_O+1+N;      //the central pixel
            mascara<=mask[3];
            estado_sig=4'b0110;
            end

4'b0110:begin
            p_inicioconv<=0;
            we_1<=0;
            ce_1<=1;
            p_i<=i;
            p_j<=j;
            p_DIRECC_D<=DIRECC_D;
            p_DIRECC_O<=DIRECC_O;
            dato_leido<=0;
            clk_en<=1;
            addr_o<=DIRECC_O+2+N;
            mascara<=mask[4];
```

```
            estado_sig=4'b0111;
            end

4'b0111:begin
            p_inicioconv<=0;
            we_1<=0;
            ce_1<=1;
            p_i<=i;
            p_j<=j;
            p_DIRECC_D<=DIRECC_D;
            p_DIRECC_O<=DIRECC_O;
            dato_leido<=0;
            clk_en<=1;
            addr_o<=DIRECC_O+N+N;
            mascara<=mask[5];
            estado_sig=4'b1000;
            end

4'b1000:begin
            p_inicioconv<=0;
            we_1<=0;
            ce_1<=1;
            p_i<=i;
            p_j<=j;
            p_DIRECC_D<=DIRECC_D;
            p_DIRECC_O<=DIRECC_O;
            dato_leido<=0;
            clk_en<=1;
            addr_o<=DIRECC_O+N+N+1;
            mascara<=mask[6];
            estado_sig=4'b1001;
            end

 4'b1001:begin
            p_inicioconv<=0;
            we_1<=0;
            ce_1<=1;
            p_i<=i;
            p_j<=j;
            p_DIRECC_D<=DIRECC_D;
            p_DIRECC_O<=DIRECC_O;
            dato_leido<=0;
            clk_en<=1;
            addr_o<=DIRECC_O+N+N+2;
            mascara<=mask[7];
            estado_sig=4'b1010;
             end

 4'b1010:begin
            p_inicioconv<=0;
             ce_1<=1;
            dato_leido<=0;
            clk_en<=1;
            mascara<=mask[8];
             p_i<=i;
             p_j<=j;
             p_DIRECC_O<=DIRECC_O;
             p_DIRECC_D<=DIRECC_D;
             if(res_listo==1'b1)
                    begin
                     clk_en<=0;
                      if(i<M-2)
                          begin
                          we_1<=1;
                          addr_o<=DIRECC_D;
                          p_i<=i+1;
                          p_j<=j;
                          p_DIRECC_D<=DIRECC_D;
                          p_DIRECC_O<=DIRECC_O+1;
                          estado_sig=4'b1011;
                          end
                     else if(j<N-2)
                          begin
                          we_1<=1;
                          addr_o<=DIRECC_D;
                          p_i<=1;
```

```
                    p_j<=j+1;
                    p_DIRECC_D<=DIRECC_D;
                    p_DIRECC_O<=DIRECC_O+3;
                    estado_sig=4'b1100;
                    end
                else
                    begin
                    we_1<=1;
                    addr_o<=DIRECC_D;
                    p_i<=1;
                    p_j<=1;
                    p_DIRECC_D<=DIRECC_D;
                    p_DIRECC_O<=0;
                    estado_sig=4'b1101;
                    end
                end
                else       estado_sig=4'b1010;
            end

        4'b1011:begin
                addr_o<=0;
                p_inicioconv<=0;
                we_1<=0;
                ce_1<=0;
                mascara<=0;
                p_i<=i;
                p_j<=j;
                p_DIRECC_O<=DIRECC_O;
                p_DIRECC_D<=DIRECC_D+1;
                dato_leido<=1;
                clk_en<=0;
                estado_sig=4'b0001;
                end

        4'b1100:begin
                addr_o<=0;
                p_inicioconv<=0;
                we_1<=0;
                ce_1<=0;
                mascara<=0;
                p_i<=i;
                p_j<=j;
                p_DIRECC_O<=DIRECC_O;
                p_DIRECC_D<=DIRECC_D+3;
                dato_leido<=1;
                clk_en<=0;
                estado_sig=4'b0001;
                end

        4'b1101:begin
                addr_o<=0;
                p_inicioconv<=0;
                we_1<=0;
                ce_1<=0;
                mascara<=0;
                p_i<=i;
                p_j<=j;
                p_DIRECC_O<=DIRECC_O;
                p_DIRECC_D<=0;
                clk_en<=0;
                dato_leido<=1;
                estado_sig=4'b0000;
                end

        default  estado_sig=4'bxxxx;


    endcase
  end

endmodule
```

Sumador.v

```
//////////////////////////////////////////////////////////////////////////////////////////////////
// Engineer:  Jose Fernandez Perez
//
// Create Date:    19:44:19 03/28/2006
// Module Name:    sumador.v
// Project Name:   Aquarius
// Target Devices:  X2CVP30
// Tool versions:  Xilinx ISE v8.1
// Description:   This function calculates the sum of two source images
// Revision 0.01 - File Created
//
//////////////////////////////////////////////////////////////////////////////////////////////////
module sumador(clk,rst,cesum,we,din,datin,address,we_1,cs_1,addr,dout);
    input clk;
    input rst;
    input cesum;
    input we;                                   //Aquarius wishbone signal
    input [7:0] din;                            //memory data
    input [15:0] datin;
    input [3:0] address;
    output we_1;
    output cs_1;
    output [15:0] addr;
    output [7:0] dout;

    reg [15:0] addr;
    reg [7:0] dout;
    reg cs_1;
    reg we_1;

    reg [7:0] M;
    reg [7:0] N;

    reg [15:0] DA;
    reg [15:0] DB;

    reg [15:0] p_DA;
    reg [15:0] p_DB;

    reg [15:0] DO1;
    reg [15:0] DO2;
    reg [15:0] DD;
    reg iniciosum;
    reg [7:0] i;
    reg [7:0] j;

    reg [2:0] estado_act;
    reg [2:0] estado_sig;

    reg [15:0] p_DD;
    reg [15:0] p_DO1;
    reg [15:0] p_DO2;
    reg p_iniciosum;
    reg [7:0] p_i;
    reg [7:0] p_j;

always @(posedge clk or posedge rst)
    begin
        if(rst)
            begin
                M<=0;
                N<=0;
                DO1<=0;
                DO2<=0;
                DA<=0;
                DB<=0;
                DD<=0;
                iniciosum<=0;
                i<=0;
                j<=0;
                estado_act<=3'b000;
            end
        else
```

```
            begin
              estado_act<=estado_sig;
              DO1<=p_DO1;
              DO2<=p_DO2;
              DD<=p_DD;
              DA<=p_DA;
              DB<=p_DB;
              iniciosum<=p_iniciosum;
              i<=p_i;
              j<=p_j;
              if(cesum==1'b1 && we==1'b1)     //cesum*we
                 begin
                      if(address==4'b0000) M<=datin[7:0];
                      else if(address==4'b0001) N<=datin[7:0];
                      else if(address==4'b0010) DO1<=datin;
                      else if(address==4'b0100) DO2<=datin;
                      else if(address==4'b0110) DD<=datin;
                      else if(address==4'b1000) iniciosum<=datin[0];
                  end
            end

      end


always @(estado_act or M or N or din or DA or DB or DO1 or DO2 or DD or iniciosum or i or j)
    begin
      case (estado_act)
            3'b000:begin
                      p_DO1<=DO1;
                      p_DO2<=DO2;
                      p_DD<=DD;
                      p_iniciosum<=iniciosum;
                      cs_1<=0;
                      we_1<=0;
                      addr<=0;
                      dout<=0;
                      p_i<=0;
                      p_j<=0;
                      p_DA<=DA;
                      p_DB<=DB;
                      if(iniciosum==1'b1)
                         estado_sig=3'b001;
                      else
                      estado_sig=3'b000;
                      end

             3'b001:begin
                      p_DO1<=DO1;
                      p_DO2<=DO2;
                      p_DD<=DD;
                      p_iniciosum<=0;
                      p_i<=i;
                      p_j<=j;
                      cs_1<=1;
                      we_1<=0;
                      p_DA<=DA;
                      p_DB<=DB;
                      dout<=0;
                      addr<=DO1;
                      estado_sig=3'b010;
                      end

             3'b010:begin
                      p_DO1<=DO1;
                      p_DO2<=DO2;
                      p_DD<=DD;
                     p_iniciosum<=0;
                      p_i<=i;
                      p_j<=j;
                      cs_1<=1;
                      we_1<=0;
                      p_DA<=din;
                      p_DB<=DB;
                      dout<=0;
                      addr<=DO2;
                      estado_sig=3'b011;
```

```
                end

            3'b011:begin
                p_DO1<=DO1;
                p_DO2<=DO2;
                p_DD<=DD;
                p_iniciosum<=0;
                p_i<=i;
                p_j<=j;
                cs_1<=1;
                we_1<=0;
                addr<=DO2;
                dout<=0;
                p_DA<=DA;
                p_DB<=din;
                estado_sig=3'b100;
                end

            3'b100:begin
                p_iniciosum<=0;
                cs_1<=1;
                we_1<=1;
                addr<=DD;
                p_DA<=DA;
                p_DB<=DB;
                if((DA+DB)>255)
                   dout<=255;                          //positive saturation
                else
                   dout<=DA+DB;
                p_DO1<=DO1+1;
                p_DO2<=DO2+1;
                p_DD<=DD+1;
                if(i<M-1)
                      begin
                          p_i<=i+1;
                          p_j<=j;
                          estado_sig=3'b001;
                      end
                else if(j<N-1)
                      begin
                          p_i<=0;
                          p_j<=j+1;
                          estado_sig=3'b001;
                      end
                else
                      begin
                          p_i<=0;
                          p_j<=0;
                          estado_sig=3'b000;
                      end
                end

            default estado_sig=3'bxxx;
        endcase

    end

endmodule
```

Restador.v

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Engineer:  Jose Fernandez Perez
//
// Create Date:    19:44:19 03/28/2006
// Module Name:    restador.v
// Project Name:   Aquarius
// Target Devices:  X2CVP30
// Tool versions:  Xilinx ISE v8.1
// Description:   This function calculates the difference between two source images
// Revision 0.01 - File Created
//
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module restador(clk,rst,ceres,we,din,datin,address,we_1,cs_1,addr,dout);
    input clk;
    input rst;
    input ceres;
    input we;                                       //Aquarius wishbone signal
    input [7:0] din;                                //memory data
    input [15:0] datin;
    input [3:0] address;
    output we_1;
    output cs_1;
    output [15:0] addr;
    output [7:0] dout;

    reg [15:0] addr;
    reg [7:0] dout;
    reg cs_1;
    reg we_1;
    reg [7:0] M;
    reg [7:0] N;

    reg [15:0] DA;
    reg [15:0] DB;
    reg [15:0] p_DA;
    reg [15:0] p_DB;

    reg [15:0] DO1;
    reg [15:0] DO2;
    reg [15:0] DD;
    reg iniciores;
    reg [7:0] i;
    reg [7:0] j;

    wire signed[15:0] cuenta;

    reg [2:0] estado_act;
    reg [2:0] estado_sig;

    reg [15:0] p_DD;
    reg [15:0] p_DO1;
    reg [15:0] p_DO2;
    reg p_iniciores;
    reg [7:0] p_i;
    reg [7:0] p_j;


    assign cuenta = 128 + DA - DB;                  //the output value without saturation

always @(posedge clk or posedge rst)
    begin
            if(rst)
                begin
                M<=0;
                N<=0;
                DO1<=0;
                DO2<=0;
                 DA<=0;
                 DB<=0;
                DD<=0;
                iniciores<=0;
                i<=0;
                j<=0;
```

```
                estado_act<=3'b000;
                end
            else
              begin
                estado_act<=estado_sig;
                DO1<=p_DO1;
                DO2<=p_DO2;
                DD<=p_DD;
                DA<=p_DA;
                DB<=p_DB;
                iniciores<=p_iniciores;
                i<=p_i;
                j<=p_j;
                if(ceres==1'b1 && we==1'b1)     //ceres*we
                  begin
                        if(address==4'b0000) M<=datin[7:0];
                        else if(address==4'b0001) N<=datin[7:0];
                        else if(address==4'b0010) DO1<=datin;
                        else if(address==4'b0100) DO2<=datin;
                        else if(address==4'b0110) DD<=datin;
                        else if(address==4'b1000) iniciores<=datin[0];
                  end
                end

      end

always @(estado_act or M or N or din or DA or DB or DO1 or DO2 or DD or iniciores or i or j or cuenta)
    begin
      case (estado_act)
            3'b000:begin
                        p_DO1<=DO1;
                        p_DO2<=DO2;
                        p_DD<=DD;
                        p_iniciores<=iniciores;
                        cs_1<=0;
                        we_1<=0;
                        addr<=0;
                        dout<=0;
                        p_i<=0;
                        p_j<=0;
                        p_DA<=DA;
                        p_DB<=DB;
                        if(iniciores==1'b1)
                             estado_sig=3'b001;
                        else
                             estado_sig=3'b000;
                        end

             3'b001:begin
                        p_DO1<=DO1;
                        p_DO2<=DO2;
                        p_DD<=DD;
                        p_iniciores<=0;
                        p_i<=i;
                        p_j<=j;
                        cs_1<=1;
                        we_1<=0;
                        p_DA<=DA;
                        p_DB<=DB;
                        dout<=0;
                        addr<=DO1;
                        estado_sig=3'b010;
                        end

             3'b010:begin
                        p_DO1<=DO1;
                        p_DO2<=DO2;
                        p_DD<=DD;
                       p_iniciores<=0;
                        p_i<=i;
                        p_j<=j;
                        cs_1<=1;
                        we_1<=0;
                        p_DA<=din;
                        p_DB<=DB;
                        dout<=0;
```

```
                addr<=DO2;
                estado_sig=3'b011;
                end

        3'b011:begin
                p_DO1<=DO1;
                p_DO2<=DO2;
                p_DD<=DD;
                p_iniciores<=0;
                p_i<=i;
                p_j<=j;
                cs_1<=1;
                we_1<=0;
                addr<=DO2;
                dout<=0;
                p_DA<=DA;
                p_DB<=din;
                estado_sig=3'b100;
                end

        3'b100:begin
                p_iniciores<=0;
                cs_1<=1;
                we_1<=1;
                addr<=DD;
                p_DA<=DA;
                p_DB<=DB;
                if(cuenta>255)
                   dout<=255;                    //positive saturation
                else if(cuenta<0)
                  dout<=0;                       //negative saturation
                else
                   dout<=cuenta;
                p_DO1<=DO1+1;
                p_DO2<=DO2+1;
                p_DD<=DD+1;
                if(i<M-1)
                  begin
                      p_i<=i+1;
                      p_j<=j;
                      estado_sig=3'b001;
                  end
                else if(j<N-1)
                  begin
                      p_i<=0;
                      p_j<=j+1;
                      estado_sig=3'b001;
                  end
                else
                  begin
                      p_i<=0;
                      p_j<=0;
                      estado_sig=3'b000;
                   end
                end

         default estado_sig=3'bxxx;

      endcase

   end

endmodule
```

Mac.v

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////
// Engineer:   Jose Fernandez Perez
//
// Create Date:    19:44:19 03/28/2006
// Module Name:  mac.v
// Project Name:  Aquarius
// Target Devices: X2CVP30
// Tool versions: Xilinx ISE v8.1
// Description: This function computes the sum of products that takes place
//                    on a spatial masks based processing
// Revision 0.01 - File Created
//
//////////////////////////////////////////////////////////////////////////////////////////////////////////
module mac(dataa,datab,clk,rst,aclr,clken,dataout,res_listo);
   input signed[7:0] dataa;
   input signed [7:0] datab;
   input clk;
   input rst;
   input aclr;
   input clken;
   output signed [16:0] dataout;
   output res_listo;



   reg signed [16:0] dataout;
   reg res_listo;
   reg signed[8:0] dataa_reg;
   reg signed[7:0] datab_reg;
   reg signed[16:0] multa_reg;
   reg [3:0] count;
   reg [7:0] PIX;

   wire signed[16:0] salida;
   wire signed[16:0] salida_2;

   wire signed[16:0] multa;
   wire signed[16:0] adder_out;
   wire signed[16:0] adder_out2;

   assign salida = adder_out/32 + PIX;                    //the output value without saturation
   assign salida_2 = adder_out2/32 - PIX;                 //this equations are used to simulate
   assign multa = dataa_reg * datab_reg;                  //a CNN operation
   assign adder_out = multa_reg + dataout;
   assign adder_out2 = - adder_out;

always @(posedge clk or posedge aclr or posedge rst)
   begin

      if (rst)
        begin
            dataa_reg<=0;
            datab_reg<=0;
            multa_reg<=0;
            dataout<=0;
            count<=0;
            res_listo<=0;
            PIX<=0;
        end

      else if (aclr)
        begin
            dataa_reg<=0;
            datab_reg<=0;
            multa_reg<=0;
            dataout<=0;
            count<=0;
            res_listo<=0;
            PIX<=0;
        end

      else if (clken)
        begin
```

```
            dataa_reg[7:0]<=dataa[7:0];
            dataa_reg[8]<=0;
            datab_reg<=datab;
            multa_reg<=multa;
            count<=count+1;
            if(count==5)
                  begin
                        PIX<=dataa[7:0];
                        dataout<=adder_out;
                   end
            else if(count==11)
                  begin
                        if(adder_out>=0)
                          begin
                            if((salida<256) && (salida>=0))
                                dataout <= salida;
                            else
                                dataout <= 255;                         //positive saturation
                          end
                        else
                          begin
                            if((salida_2<=-1) && (salida_2>-256))
                              dataout <= - salida_2;
                            else
                                dataout <= 0;                           //negative saturation
                          end
                  res_listo<=1;
                  end
            else
                  begin
                     dataout<=adder_out;
                  end
            end

      end
endmodule
```

Threshold.v

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Engineer: Jose Fernandez Perez
//
// Create Date:    19:44:19 03/28/2006
// Module Name:    threshold.v
// Project Name:   Aquarius
// Target Devices:  X2CVP30
// Tool versions:  Xilinx ISE v8.1
// Description:   This function calculates a binary image by comparing
//                the gray-level values of a source image with a threshold value
//
// Revision 0.01 - File Created
//
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module threshold(datin,addr_i,clk,rst,din,cethr,we,dout,addr_o,we_1,cs_1);
    input [15:0] datin;
    input [2:0] addr_i;
    input clk;
    input rst;
    input [7:0] din;
    input cethr;
    input we;                                          //Aquarius wishbone signal
    output [7:0] dout;
    output [15:0] addr_o;
    output we_1;
    output cs_1;


    reg [15:0] addr_o;
    reg [7:0] dout;
    reg we_1;
    reg cs_1;

    reg [7:0] M;
    reg [7:0] N;
    reg [15:0] DO;
    reg [15:0] DD;
    reg iniciothr;

    reg [7:0] T;

    reg [7:0] DAT;
    reg [7:0] p_DAT;

    reg [7:0] i;
    reg [7:0] j;

    reg [1:0] estado_act;
    reg [1:0] estado_sig;

    reg [15:0] p_DO;
    reg [15:0] p_DD;
    reg p_iniciothr;
    reg [7:0] p_i;
    reg [7:0] p_j;

always @(posedge clk or posedge rst)
    begin
      if(rst)
            begin
                M<=0;
                N<=0;
                DO<=0;
                DD<=0;
                iniciothr<=0;
                DAT<=0;
                i<=0;
                j<=0;
                T<=90;
               estado_act<=2'b00;
            end
      else
            begin
```

```
                estado_act<=estado_sig;
                DO<=p_DO;
                DD<=p_DD;
                iniciothr<=p_iniciothr;
                i<=p_i;
                j<=p_j;
                DAT<=p_DAT;
                if(cethr==1'b1 && we==1'b1)     //cethr*we
                   begin
                        if(addr_i==3'b000) M<=datin[7:0];
                         else if(addr_i==3'b001) N<=datin[7:0];
                         else if(addr_i==3'b010) DO<=datin;
                         else if(addr_i==3'b100) DD<=datin;
                         else if(addr_i==3'b110) iniciothr<=datin[0];
                         else if(addr_i==3'b111) T<=datin[7:0];                          //the threshold level
                   end
                end

        end


always @(estado_act or M or N or din or DAT or DO or DD or iniciothr or i or j or T)
     begin
        case (estado_act)
                 2'b00:begin
                        p_DO<=DO;
                        p_DD<=DD;
                        p_iniciothr<=iniciothr;
                        cs_1<=0;
                        we_1<=0;
                        addr_o<=0;
                        dout<=0;
                        p_DAT<=0;
                        p_i<=0;
                        p_j<=0;
                        if(iniciothr==1'b1)
                           estado_sig=2'b01;
                        else
                           estado_sig=2'b00;
                        end

                 2'b01:begin
                        p_DO<=DO;
                        p_DD<=DD;
                        p_iniciothr<=0;
                        p_DAT<=0;
                        p_i<=i;
                        p_j<=j;
                        cs_1<=1;
                        we_1<=0;
                        dout<=0;
                        addr_o<=DO;
                        estado_sig=2'b10;
                        end

                 2'b10:begin
                        p_DO<=DO;
                        p_DD<=DD;
                        p_iniciothr<=0;
                        p_i<=i;
                        p_j<=j;
                        cs_1<=1;
                        we_1<=0;
                        p_DAT<=din;
                        dout<=0;
                        addr_o<=DO;
                        estado_sig=2'b11;
                        end

                  2'b11:begin
                        p_iniciothr<=0;
                        cs_1<=1;
                        we_1<=1;
                        addr_o<=DD;
                        if(DAT<=T)
                          begin
```

```
                    dout<=0;
                  end
                else
                  begin
                    dout<=255;
                  end
                p_DO<=DO+1;
                p_DD<=DD+1;
                if(i<M-1)
                    begin
                        p_i<=i+1;
                        p_j<=j;
                        estado_sig=2'b01;
                    end
                else if(j<N-1)
                    begin
                        p_i<=0;
                        p_j<=j+1;
                        estado_sig=2'b01;
                    end
                else
                    begin
                        p_i<=0;
                        p_j<=0;
                        estado_sig=2'b00;
                    end
                end

            default estado_sig=2'bxx;

        endcase

    end

endmodule
```

8.2 Códigos C

El código que se muestra a continuación corresponde al diagrama de flujo que aparecía en la figura 5.1.

```
#include "common.h"

//******************************************************
//*
//*       PROTOTIPOS DE FUNCIONES UART
//*
//******************************************************

void         uart_tx(unsigned char data);
unsigned char uart_rx(void);
unsigned char uart_rx_echo(void);
void         uart_rx_flush(void);
void         uart_set_baudrate(void);


//******************************************************
//*
//*       PROGRAMA PRINCIPAL
//*
//******************************************************

int main_sh()
{

   int num;
   int j;
   int i;
   unsigned char dummy;

   //====================================
   // INICIALIZACIÓN MEMORIA
   //====================================

   j=4096;
   for(j=4096;j<12288;j++) DPRAM.MEMO[j]=0;

   //====================================
   // INICIALIZACIÓN UART
   //====================================

   uart_set_baudrate();
   uart_rx_flush();


   //====================================
   // CONFIGURACIÓN CONVOLUCIONADOR
   //====================================

   CONV.MASK0=0;
   CONV.MASK1=1;
   CONV.MASK2=0;
   CONV.MASK3=1;
   CONV.MASK4=-4;
   CONV.MASK5=1;
   CONV.MASK6=0;
   CONV.MASK7=1;
   CONV.MASK8=0;
   CONV.M=64;
   CONV.N=64;

   //====================================
   // CONFIGURACIÓN UMBRALIZADOR
   //====================================

   THR.M=64;
   THR.N=64;
   THR.T=90;

   //====================================
```

```
  // CONFIGURACIÓN RESTADOR
  //====================================

  RES.M=64;
  RES.N=64;


  //====================================
  // BUCLE PRINCIPAL
  //====================================

while(1)
    {
      num=0;
      j=0;


       //====================================
       // Recepción imagen original
       //====================================
       while(num<4096)
        {
         DPRAM.MEMO[num]=uart_rx();
         num=num+1;
        }

    //====================================
    // PARÁMETROS 1ºCONVOLUCIÓN
    //====================================
    CONV.DO=0;
    CONV.DD=4096;

    CONV.INICIOCONV=1;                          /*inicio de convolución*/
    for(j=0;j<30000;j++) dummy = PORTI.RESERVED_0;  /*bucle de espera fin convolución*/


   //====================================
   // 34 CONVOLUCIONES RESTANTES
   //====================================

  for(i=0;i<17;i++)
   {
    j=0;

    CONV.DO=4096;
    CONV.DD=8192;

    CONV.INICIOCONV=1;
    for(j=0;j<30000;j++) dummy = PORTI.RESERVED_0;

    j=0;

    CONV.DO=8192;
    CONV.DD=4096;

    CONV.INICIOCONV=1;
    for(j=0;j<30000;j++) dummy = PORTI.RESERVED_0;
   }

   //================================================
   // Envío 1º imagen procesada (35 convoluciones)
   //================================================

    num=4096;

       while(num<8192)
         {
           uart_tx(DPRAM.MEMO[num]);
           num=num+1;
         }

    dummy=uart_rx();            /*protocolo con Visual Basic*/

   //====================================
   // CONVOLUCIÓN SIMPLE
   //====================================
```

```
j=0;

CONV.DO=0;
CONV.DD=8192;

CONV.INICIOCONV=1;
for(j=0;j<13455;j++);

//=========================================
// Envío 2º imagen procesada (1 convolución)
//=========================================

 num=8192;

   while(num<12288)
    {
      uart_tx(DPRAM.MEMO[num]);
      num=num+1;
    }

dummy=uart_rx();            /*protocolo con Visual Basic*/


//=========================================
// PARÁMETROS RESTA DE IMÁGENES PROCESADAS
//=========================================

j=0;

RES.DO1=4096;
RES.DO2=8192;
RES.DD=12288;

RES.INICIORES=1;                                        /*inicio de resta*/
for(j=0;j<10000;j++) dummy = PORTI.RESERVED_0;

//====================================
// Envío de imagen diferencia
//====================================

 num=12288;

   while(num<16384)
    {
      uart_tx(DPRAM.MEMO[num]);
      num=num+1;
    }


dummy=uart_rx();          /*protocolo con Visual Basic*/


//============================
// PARÁMETROS UMBRALIZADO
//============================

j=0;

THR.DO=12288;
THR.DD=16384;

THR.INICIOTHR=1;                                        /*inicio de umbralizado*/
for(j=0;j<10000;j++) dummy = PORTI.RESERVED_0;


//============================
// Envío imagen umbralizada
//============================

num=16384;

   while(num<20480)
    {
      uart_tx(DPRAM.MEMO[num]);
      num=num+1;
```

```
        }

    }

 return 0;
}



//*****************************************************
//*
//*       Funciones de manejo de UART
//*
//*****************************************************

//================================
// Envía Tx
//================================
void uart_tx(unsigned char data)
{
   while(UART.UARTCON.BIT.TXF);
   UART.BYTE.TX = data;
}

//====================================
// Recibe RX
//====================================
unsigned char uart_rx(void)
{
   while(UART.UARTCON.BIT.RXE);
   return(UART.BYTE.RX);
}

//=========================================
// Recibe RX con eco a TX
//=========================================
unsigned char uart_rx_echo(void)
{
   unsigned char data;

   while(UART.UARTCON.BIT.RXE);
   data = UART.BYTE.RX;

   while(UART.UARTCON.BIT.TXF);
   UART.BYTE.TX = data;

   return(data);
}

//==================
// Limpia FIFO RxD
//==================
void uart_rx_flush(void)
{
   unsigned char dummy;

   while(UART.UARTCON.BIT.RXE == 0) dummy = UART.BYTE.RX;
}

//==================================
// Establece Tasa Baudios 9600bps
//==================================
void uart_set_baudrate(void)
{

   int i;
   unsigned char dummy;

   UART.UARTBG0 = 18;
   UART.UARTBG1 = 65;

   for (i = 0; i < 65536; i++) dummy = PORTI.RESERVED_0;
}
```

8.3 Código Visual Basic

Código que gestiona la interfaz entre el sistema de test y desarrollo implementado en la placa y el PC encargado de controlar el sistema. Presenta una interfaz al usuario de la forma mostrada en la figura 4.6.

```
Imports System
Imports System.IO
Imports System.ComponentModel
Imports System.Threading
Imports System.Windows.Forms

Delegate Sub SetTextCallback(ByVal Recibido As String)
Delegate Sub SetDisplayCallback(ByVal imagen() As Byte)

Public Class Form1
    Dim N_IMAGEN As Integer = 1
    Dim IMAGEN(5, 4096) As Byte

    Dim SerialPort1 As New System.IO.Ports.SerialPort("COM1", 9600, IO.Ports.Parity.None,
8, IO.Ports.StopBits.One)

    Private Sub SetText(ByVal Recibido As String)
        'InvokeRequired required compares the thread ID of the
        'calling thread to the thread ID of the creating thread.
        'If these threads are different, it returns true.
        If Me.TextBox3.InvokeRequired Then
            Dim d As New SetTextCallback(AddressOf SetText)
            Me.Invoke(d, New Object() {Recibido})
        Else
            Me.TextBox3.AppendText(vbNewLine & Recibido)
        End If
    End Sub

    Private Sub SetDisplay(ByVal imagen() As Byte)
        'InvokeRequired required compares the thread ID of the
        'calling thread to the thread ID of the creating thread.
        'If these threads are different, it returns true.
        If Me.Display1.InvokeRequired Then
            Dim d As New SetDisplayCallback(AddressOf SetDisplay)
            Me.Invoke(d, New Object() {imagen})
        Else
            Me.Display1.escribe(imagen)
        End If
    End Sub



    Private Sub SerialPort1_DataReceived(ByVal sender As System.Object, ByVal e As
System.IO.Ports.SerialDataReceivedEventArgs)
        Dim Recibido(4096) As Byte
        Dim Leidos As Integer
        Dim dummy(2) As Byte
        Dim OK As Boolean = True
        Dim sAr As String
        Dim sw As IO.StreamWriter
        Dim AppPath As String = IO.Directory.GetCurrentDirectory


        Leidos = SerialPort1.BytesToRead

        If (Leidos = 4096) Then
            sAr = AppPath & "\" & TextBox2.Text & N_IMAGEN.ToString & ".txt"
            SetText(">> Recibida imagen " & N_IMAGEN.ToString)
            Try
                sw = New IO.StreamWriter(sAr)
            Catch ex As Exception
                SetText(">> Se produjo el siguinte error al abrir el archivo " &
                        "\recibido" & N_IMAGEN.ToString & ".txt")
                SetText(">> " & ex.Message.ToString)
                OK = False
            End Try

            SerialPort1.Read(Recibido, 0, Leidos)
```

```
            Dim i As Integer = 0
            For i = 0 To 4095
                IMAGEN(N_IMAGEN, i) = Recibido(i)
            Next

            SetDisplay(Recibido)

            For i = 0 To (Leidos - 1)
                Try
                    sw.WriteLine(Recibido(i))
                Catch ex As Exception
                    SetText(">> Error de escritura en " & "\recibido" & N_IMAGEN.ToString &
                            ".txt")
                    SetText(">> " & ex.Message.ToString)
                    OK = False
                End Try
            Next

            If (OK) Then
                SetText(">> Se ha guardado correctamente la imagen " & N_IMAGEN.ToString)
            Else
                SetText(">> ERROR DE ESCRITURA: La imagen " & N_IMAGEN.ToString & " no se
                        guardó")
            End If

            ProgressBar1.Increment(23)

            If (N_IMAGEN < 4) Then
                N_IMAGEN = N_IMAGEN + 1
                dummy(0) = 114
                SerialPort1.Write(dummy, 0, 1)
                SetText(">> Enviada a la FPGA petición para nueva imagen")
            Else
                SetText(">> TODAS LAS IMÁGENES RECIBIDAS. PROCESO TERMINADO")
                GroupBox2.Enabled = True
                GroupBox1.Enabled = True
                RadioButton1.Checked = False
                RadioButton2.Checked = False
                RadioButton3.Checked = False
                RadioButton4.Checked = False
                RadioButton5.Checked = False
                EnviarButton.Text = "Comenzar proceso"
                ProgressBar1.Value = 0

            End If

            sw.Close()
        End If

    End Sub

    Private Sub EnviarButton_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles EnviarButton.Click

        Dim Enviada(4096) As Byte
        Dim OK As Boolean = True
        Dim sAr As String
        Dim sr As IO.StreamReader
        Dim AppPath As String = IO.Directory.GetCurrentDirectory

        GroupBox2.Enabled = False
        GroupBox1.Enabled = False
        EnviarButton.Text = "Procesando..."
        ProgressBar1.Value = 0

        N_IMAGEN = 1
        sAr = AppPath & "\" & TextBox1.Text.ToString

        Try
            sr = New IO.StreamReader(sAr)
        Catch ex As Exception
            TextBox3.AppendText(vbNewLine & ">> Se produjo el siguinte error al abrir el
                                archivo " & TextBox1.Text.ToString)
            TextBox3.AppendText(vbNewLine & ">> " & ex.Message.ToString)
            OK = False
        End Try
```

```
        Dim i As Integer
        Dim s As String

        If (OK) Then
            For i = 0 To 4095
                s = sr.ReadLine
                Enviada(i) = CByte(Val(s))
                IMAGEN(0, i) = Enviada(i)
            Next

            Display1.escribe(Enviada)
            TextBox3.AppendText(vbNewLine & ">> Leida imagen original")
            ProgressBar1.Increment(5)

            SerialPort1.Write(Enviada, 0, 4096)
            TextBox3.AppendText(vbNewLine & ">> Se envió la imagen original a la FPGA")
            sr.Close()
        End If



    End Sub

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles MyBase.Load
        Try
            SerialPort1.Open()
            AddHandler SerialPort1.DataReceived, AddressOf SerialPort1_DataReceived
        Catch
            TextBox3.AppendText(">> Se Produjo un error al abrir el puerto COM1. Por favor,
                                revise el HW.")
        End Try

        If SerialPort1.IsOpen = True Then
            TextBox3.AppendText(">> Puerto COM1 abierto.")
        End If

    End Sub

    Private Sub Form1_FormClosing(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.FormClosingEventArgs) Handles MyBase.FormClosing
        SerialPort1.Close()
    End Sub

    Private Sub TextBox3_TextChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles TextBox3.TextChanged

    End Sub

    Private Sub RadioButton1_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles RadioButton1.CheckedChanged
        Dim imagenlocal(4096) As Byte
        Dim i As Integer = 0

        If RadioButton1.Checked = True Then
            For i = 0 To 4095
                imagenlocal(i) = IMAGEN(0, i)
            Next
            Display1.escribe(imagenlocal)
        End If
    End Sub

    Private Sub RadioButton2_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles RadioButton2.CheckedChanged
        Dim imagenlocal(4096) As Byte
        Dim i As Integer = 0

        If RadioButton2.Checked = True Then
            For i = 0 To 4095
                imagenlocal(i) = IMAGEN(1, i)
            Next
            Display1.escribe(imagenlocal)
        End If
    End Sub
```

```
    Private Sub RadioButton3_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles RadioButton3.CheckedChanged
        Dim imagenlocal(4096) As Byte
        Dim i As Integer = 0

        If RadioButton3.Checked = True Then
            For i = 0 To 4095
                imagenlocal(i) = IMAGEN(2, i)
            Next
            Display1.escribe(imagenlocal)
        End If
    End Sub

    Private Sub RadioButton4_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles RadioButton4.CheckedChanged
        Dim imagenlocal(4096) As Byte
        Dim i As Integer = 0

        If RadioButton4.Checked = True Then
            For i = 0 To 4095
                imagenlocal(i) = IMAGEN(3, i)
            Next
            Display1.escribe(imagenlocal)
        End If
    End Sub

    Private Sub RadioButton5_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles RadioButton5.CheckedChanged
        Dim imagenlocal(4096) As Byte
        Dim i As Integer = 0

        If RadioButton5.Checked = True Then
            For i = 0 To 4095
                imagenlocal(i) = IMAGEN(4, i)
            Next
            Display1.escribe(imagenlocal)
        End If
    End Sub
End Class
```

8.4 Código Matlab

El código que se muestra a continuación muestra una función que compone una imagen en un formato adecuado para su transmisión a la FPGA.

```
function procesa2(A)

% Funcion encargada de generar sentencias en formato adecuado para transmitir a FPGA
% partiendo de una imagen A
    i=1;
    j=1;
    X=double(A);                        %cambio de formato para el manejo de píxeles
    [M,N]=size(X);

    FD=fopen('enviar.txt','w');

    for i=1:M
       for j=1:N
            fprintf(FD,'%d \n',X(i,j));          %paso de la imagen a un fichero
         end
     end

     fclose('all');
```