PROCESAMIENTO DIGITAL DE IMÁGENES MÉDICAS SOBRE PLATAFORMAS FPGAs

Tesis de Maestría presentada por

Jorge Osio

ante la

Facultad de Ingeniería de la

Universidad Nacional de La Plata

para la obtención del grado académico de

MAGISTER EN INGENIERIA

Dirección de tesis:

Director: Antonio Adrian Quijano

Co-director: José Antonio Rapallini

La Plata, Diciembre de 2018

Centro de Técnicas Analógico-Digitales UIDET CeTAD

Depto. de Electrotecnia, Facultad de Ingeniería, UNLP

Resumen

Los equipos de imágenes médicas cobran un papel progresivamente crítico en el cuidado de la salud. La predicción y el tratamiento se manejan combinando modalidades tales como la tomografía por emisión de positrones (PET), Tomografía computarizada (CT) y equipos de rayos X. Se requieren sistemas sofisticados de software/hardware para el análisis de señales que posibilitan la obtención y procesamiento de imágenes de altas resoluciones. Estos sistemas deben proveer un procesamiento muy exacto y sumamente rápido de grandes cantidades de datos de imágenes. Estos requerimientos en sistemas que a su vez deben poseer la capacidad de actualizar continuamente sus características y algoritmos, no deja otra alternativa que usar sistemas programables como CPUs y FPGAs potentes.

En este trabajo se desarrolló un sistema en el que se busca optimizar la eficiencia en la ejecución de algoritmos de procesamiento digital de imágenes sobre dispositivos FPGAs, aprovechando las bondades que proveen en cuanto al cómputo paralelo y la concurrencia.

Para el procesamiento, los algoritmos se aplicaron a imágenes médicas, ya que las necesidades de desempeño para el análisis en tiempo real de las mismas, requieren plataformas de sistemas escalables tanto en software (CPUs) como en hardware (lógica configurable). Además, estas plataformas procesadoras deben encontrar un equilibrio entre desempeño/precio y deben ser capaces de procesar múltiples variedades de imágenes.

La metodología desarrollada consiste en la implementación y ejecución de algoritmos de procesamiento de imágenes en un simple procesador, luego en dos procesadores y por último en un sistema de dos procesadores con coprocesador. En este último caso, los procesadores se encargan del manejo y organización de los datos y los coprocesadores de la ejecución de la parte principal del algoritmo de forma concurrente. El método se aplica a los algoritmos de procesamiento de imágenes de tipo espacial basados en operadores de ventana, tales como el filtro de media, mediana, erosión, dilatación y la convolución espacial.

Agradecimientos

Mis más sinceros agradecimientos a toda la gente que de una forma u otra hizo que este trabajo fuera posible:

- A mis directores, Antonio y José, por su apoyo y paciencia a lo largo de todo el proceso de desarrollo y escritura del trabajo.

- A la UIDET CeTAD, que me brindó el espacio físico y los recursos materiales necesarios para el trabajo diario.

- Muy especialmente a mis compañeros de trabajo, becarios y alumnos de trabajo final, sin cuyo aporte y colaboración este trabajo no se hubiera consumado: Walter, Leonardo C., Eduardo, Leonardo N, Daniel, Federico, Franco, Luis, Martín, Marcelo, Ariel y Victoria.

- A mis Padres Miguel y Beatriz que me dieron todo para poder llegar a donde he llegado.

- A mis hermanos y a mi Familia por todo su apoyo incondicional. A mi compañera de la vida Evange por estar en los momentos difíciles.

- Por último, a mis amigos que son el cable a tierra que hace falta para desconectarse de la rutina diaria.

Índice general

Índice de figuras

Índice de cuadros

Lista de abreviaturas

PET: Tomografía por emisión de positrones

CT: Tomografía computarizada

CPU: Unidad Central de Procesamiento

FPGA: Arreglo de compuertas programables en campo

DSP: Procesamiento digital de señales

ASIC: Circuitos integrados de aplicación específica

SOFT-COREPROCESSOR: Procesador implementado por software en una FPGA

MICROBLAZE: Procesador embebidos que se implementa en las FPGAs de Xilinx

LNSI: Filtros lineales no invariantes a la traslación

LSI: Filtros lineales invariantes a la traslación

SOPC: System on prog chip, circuito integrado por diversos módulos vlsi

BLOQUES IP: Bloques Funcionales en VHDL de Propiedad Intelectual

SISD: Sistemas simple flujo de instrucción y simple flujo de datos

SIMD: Sistemas simple flujo de instrucción y múltiple flujo de datos

MISD: Sistemas múltiple flujo de instrucción y simple flujo de datos

MIMD: Sistemas múltiple flujo de instrucción y múltiple flujo de datos

SMP: Multiprocesamiento simétrico

GMSV: memoria centralizada y variables compartidas.

GMMP: memoria centralizada y paso de mensajes.

DMSV: memoria distribuida y variables compartidas.

DMMP: memoria distribuida y paso de mensajes

LMB: bus local de memoria

PLB: bus local del procesador

FSL: bus de acceso rápido

MPMC: Controlador de memoria multipuerto

1. Introducción

1.1. Estado actual del tema

Los equipos de imágenes médicas cobran un papel progresivamente crítico en el cuidado de la salud. La detección de afecciones se manejan combinando modalidades tales como la tomografía por emisión de positrones (PET), Tomografía computarizada (CT) y equipos de rayos X [1]. Se requiere geometría fina con micro-arreglos de detectores acoplados con sistemas sofisticados de software/hardware para el análisis de señales fotónicas y electrónicas que posibilitan la obtención de imágenes de altas resoluciones.

Estos sistemas deben proveer un procesamiento muy exacto y sumamente rápido de grandes cantidades de datos de imágenes. Estos requerimientos en sistemas que a su vez deben poseer la capacidad de actualizar continuamente sus características y algoritmos, cuya implementación debe ser flexible y contemplar la modalidad de fusión de métodos, no deja otra alternativa que usar sistemas programables como CPUs y FPGAs potentes [2].

Se deben tener en cuenta varios factores en el desarrollo eficiente de equipos flexibles de imágenes médicas:

-El desarrollo de algoritmos de imágenes requiere herramientas para el modelado intuitivo de alto nivel para las mejoras continuas en el procesamiento digital de señales (DSP) [2].

-Las necesidades de desempeño para el análisis en tiempo real requieren plataformas de sistemas escalables tanto en software (CPUs) como en hardware (lógica configurable). Estas plataformas procesadoras deben encontrar diversos puntos de desempeño/precio y deben ser capaces de procesar múltiples variedades de imágenes.

-Para particionar rápidamente y depurar algoritmos sobre estas plataformas, se deben usar herramientas de alto nivel y librerías reutilizables, para acelerar su implementación.

Teniendo en cuenta estos factores, es de gran importancia implementar algoritmos de procesamiento de imágenes que además de posibilitar el realce de las imágenes médicas, permitan obtener información adicional posibilitando la extracción de estructuras biológicas de dichas imágenes mediante técnicas de detección de bordes y crecimiento de regiones sobre dispositivos FPGAs [3]. Luego en base a los resultados obtenidos, se combinan los métodos de procesamiento implementados para acelerar y mejorar la eficiencia en el diagnóstico médico. La implementación de dichos algoritmos sobre imágenes de mucha resolución requiere grandes capacidades de cálculo, para esto se hace necesaria la utilización de técnicas de procesamiento paralelo.

En el procesamiento de imágenes médicas es indispensable poder determinar ciertas características de las imágenes obtenidas por los diferentes métodos antes nombrados. Más

específicamente, es necesaria la segmentación de las imágenes para aislar estructuras fisiológicas y biológicas de interés que permitan analizar sus características. Las técnicas de segmentación se pueden agrupar en tres clases: métodos basados en píxeles, métodos basados en regiones y métodos basados en bordes [4].

Los métodos basados en píxeles son sencillos de entender y de implementar, pero son los menos eficientes, ya que operan sobre un elemento a la vez y esto los hace sensibles al ruido. Por otro lado los métodos basados en continuidades y los basados en bordes implementan la segmentación desde lados opuestos: los métodos basados en bordes buscan las diferencias, mientras que los basados en continuidades buscan las similitudes.

Métodos Basados en Píxeles:

El algoritmo más común es el de umbralizado, en donde mediante un umbral se transforma una imagen de intensidad en una imagen binaria, quedando los objetos descriptos por un solo nivel de intensidad [4].

Métodos Basados en Continuidad:

Estos métodos buscan similaridad y consistencia en la obtención de unidades estructurales. Estas técnicas pueden ser muy efectivas en tareas de segmentación, pero sufren pérdida de definición en los bordes. Esto se produce porque están basadas en operaciones sobre un conjunto de píxeles (ventanas) y estos tienden a borronear las regiones en los bordes [5].

Los algoritmos más comunes que implementan estas técnicas son los que permiten implementar Filtros pasa bajos (promedio, mediana, gaussiano, etc). Muchas operaciones basadas en vecindad son muy usadas para distinguir texturas: Transformada de Fourier de segmentos pequeños, varianza local (o desviación estándar), el operador laplaciano, el operador por rango (las diferencias entre valores máximos y mínimos de píxeles en la vecindad), el operador Hurst (máximas diferencias en función de la separación del pixel), y el operador haralick (medición de distancia en determinado momento).

Métodos Basados en Bordes:

Estos métodos permiten obtener los bordes para luego agruparlos en conjuntos que corresponden a los lados de las unidades estructurales. El problema más común en este tipo de métodos es la detección de falsos bordes [4].

Los algoritmos de detección de bordes más comunes que utilizan estos métodos son [6]:

- Roberts
- Prewitt
- Sobel
- Laplaciano

Además, existen otros algoritmos de detección de bordes basados en las aproximaciones anteriores, tales como el algoritmo de Canny.

De cada tipo de imagen se puede obtener diferente información y se requieren determinados algoritmos para lograr un diagnóstico de forma rápida y eficiente. La posibilidad de combinar características de imágenes de algunas de estas aplicaciones exige una gran capacidad de procesamiento que requiere la implementación de técnicas de cómputo paralelo.

1.2. Motivación y objetivos de la tesis

Los sistemas actuales demandan mucha capacidad de procesamiento, no solo por la complejidad de los algoritmos que se deben ejecutar, sino también por las grandes cantidades de datos que se deben procesar.

Las FPGAs actuales tienen los suficientes recursos para poder implementar en ellas sistemas multiprocesadores complejos utilizando *soft-coreprocessors*, así como distintas arquitecturas de memoria y comunicaciones para este tipo de sistemas, por lo que son una plataforma ideal para evaluar este tipo de sistemas. Uno de los objetivos de esta tesis es evaluar las distintas posibilidades que ofrecen las FPGAs a la hora de diseñar e implementar sistemas multiprocesador basados en *soft-coreprocessors*. A lo largo de la tesis se presentarán las arquitecturas multiprocesador típicas y se desarrollará la implementada en la FPGA, analizando las ventajas e inconvenientes que posee y utilizando sus bondades para la ejecución eficiente de algoritmos de procesamiento de imágenes médicas.

La aplicación del sistema sobre el procesamiento de imágenes médicas se debe principalmente a que dichas imágenes son de alta resolución y poseen gran caudal de datos que requieren un alto grado de cómputo en tiempo real. Es por esto que se considera indispensable incursionar en técnicas de procesamiento que permitan realizar un aporte al diagnóstico por imágenes y obtener nuevas técnicas mediante la combinación de las conocidas.

Los objetivos del trabajo consisten en:

- Adquirir una formación en el manejo de algoritmos de procesamiento digital de imágenes, desde la simulación hasta la implementación de los métodos conocidos [4-6].
- Analizar los métodos tradicionales, (ejecución secuencial, concurrente y en paralelo), para ejecución de algoritmos de procesamiento de imágenes sobre Dispositivos FPGAs y evaluar la eficiencia de cada método.
- Implementar un nuevo método basado en los anteriores, que permita optimizar la ejecución de algoritmos de procesamiento de imágenes, específicamente los basados en operadores de ventana [5].
- Evaluar el desempeño de estos métodos y generalizar la implementación del mismo para la ejecución de los algoritmos antes mencionados.
- Determinar si existe algún factor por el cual la ejecución de determinados algoritmos de imágenes de tipo espacial se realiza de forma más eficiente que otros.

1.3. Organización de la tesis

En el capítulo 2 se describen las consideraciones generales en relación a las características de los algoritmos de procesamiento de imágenes que los hacen aptos para el cómputo paralelo y las características de las diferentes arquitecturas de sistemas multicore que se pueden implementar en una FPGA.

En el capítulo 3 se describirán las características de los algoritmos de procesamiento de imágenes a implementar y los pasos de ejecución e implementación en el sistema propuesto.

En el capítulo 4 se describe la arquitectura multicore a implementar y sus características. Además, se describen las características de la concurrencia que proveen los lenguajes de descripción de hardware, cuyos beneficios se verán plasmados en la implementación del sistema final. Sobre el final del capítulo se describirá la implementación del sistema completo que consistirá en la adquisición de la imagen, almacenamiento, particionamiento y ejecución de los algoritmos; mediante un procesador, dos procesadores y dos procesadores con el agregado de coprocesadores.

En el capítulo 5 se describen los resultados obtenidos de la implementación del sistema y la ejecución de algoritmos mediante los diferentes métodos, teniendo en cuenta recursos consumidos y tiempos de ejecución. Finalmente, en el capítulo 6 se desarrollarán las conclusiones que se generaron en base a los resultados.

2. Consideraciones generales

Los métodos antes nombrados operan directamente sobre los píxeles de una imagen, partiendo de esta característica en común se propondrá dividir el tratamiento de la imagen en dos partes, independientemente del algoritmo a implementar. La primera parte de la ejecución del algoritmo aprovecha la característica de los algoritmos basados en operadores de ventana para la distribución de datos y la segunda parte se ejecutará mediante procesamiento paralelo con el agregado de lógica digital aprovechando la posibilidad de concurrencia que proveen las FPGAs.

En este capítulo se describirán las consideraciones generales que se plantean para la implementación del sistema que permitirá cumplir con los objetivos propuestos. En primer lugar se describirán las características de los algoritmos que podrán ejecutarse y luego se hará una breve descripción de los distintos tipos de arquitecturas multiprocesador para terminar comentando las características del sistema multiprocesador que se implementará para la etapa de procesamiento paralelo.

2.1. Características paralelizables de los algoritmos de procesamiento de imágenes

Una imagen digital puede verse como una matriz de píxeles, donde los algoritmos de procesamiento de tipo espacial consisten en realizar operaciones entre los píxeles, más específicamente operaciones matriciales. Esto permite que la implementación de un algoritmo sea altamente paralelizable, haciendo del sistema multiprocesador algo ideal para la optimización en la ejecución de algoritmos de procesamiento de imágenes que tantos recursos consumen en los equipos de imágenes médicas.

La principal ventaja que poseen las imágenes para el cómputo paralelo, es que el procesamiento se puede realizar mediante operaciones entre matrices de manera independiente, luego en función de las características del algoritmo a implementar se pueden diagramar distintas formas de paralelismo.

2.1.1. Procesamiento espacial

El procesamiento espacial trabaja directamente sobre el valor del píxel, esto implica que las operaciones se realizan sobre cada elemento de la matriz directamente. Entonces, si nos enfocamos en el procesamiento basado en "operadores de ventana" que abarca una gran cantidad de algoritmos, se puede determinar que el procesamiento se realiza mediante una máscara, comúnmente de 3x3 o 5x5 elementos, con la que se va operando sobre los distintos píxeles de la imagen.

Casos de estudio

Como casos de estudio interesantes para aplicar procesamiento de imágenes se puede realizar una segunda clasificación en algoritmos lineales y no lineales [6], cuyas características se desarrollan a continuación.

2.1.2. Algoritmos no lineales

La principal característica de los métodos no lineales es que sus operaciones son irreversibles, entre ellos se encuentran los operadores morfológicos y otros algoritmos como el filtro de mediana, cuyos métodos toman un rumbo diferente al lineal. Los filtros no lineales utilizan un operador no lineal; esto quiere decir que no cumple la propiedad de superposición, ni la de proporcionalidad. En otras palabras, la salida no estará dada por la convolución ni la respuesta impulsiva de la entrada.

Los algoritmos no lineales se pueden clasificar en:

- Filtros de orden estadístico, que proveen robustez y adaptabilidad respecto a las distribuciones de probabilidad del ruido, permitiendo de esta manera conservar los detalles de la imagen. Entre los más representativos se cuenta con el filtro de mediana, el filtro de pila, filtro de orden clasificado, filtro M, filtro L, filtro R y filtro de media de estado alfa.

- Filtros morfológicos, que se basan en operaciones morfológicas que actúan sobre las características geométricas y topológicas del objeto.

- Filtros homomórficos, que procesan señales relacionadas convolucionalmente y de manera no lineal. Se utilizan por ejemplo en la identificación de manuscritos difusos.

- Filtros polinomiales, que se usan en situaciones en que los filtros lineales no proveen resultados óptimos, como por ejemplo en los canales no lineales.

- Filtros adaptativos, que analizan características de la imagen y su ruido para luego aplicar el método más conveniente. Permiten usar diferentes métodos en diferentes regiones de la imagen.

De toda esta variedad de algoritmos no lineales, se desarrollarán los estadísticos y los morfológicos.

Algoritmos de procesamiento morfológico de imágenes

El término procesamiento morfológico de imágenes se refiere a la clase de algoritmos que permiten obtener información sobre la estructura geométrica de una imagen [4]. La morfología puede ser usada sobre imágenes binarias o en escala de grises y se aplica en muchas áreas de procesamiento de imágenes, tales como esqueletización, detección de bordes, restauración y análisis de textura [5]. Un operador morfológico usa un elemento estructurado que consiste en una ventana, (generalmente de 3x3 o 5x5), que se va desplazando sobre los píxeles de la imagen para realizar las operaciones propias del algoritmo a implementar.

Cuando el elemento estructurado pasa sobre un conjunto de píxeles de la imagen, el elemento estructurado genera cambios o no en la imagen resultante. En el lugar donde el elemento estructurado genera cambios, se encuentra la estructura de interés de la imagen, que es lo que se obtiene en la imagen resultante. La Figura 1 muestra el concepto de un elemento estructurado que genera cambios sobre una estructura de interés de la imagen.

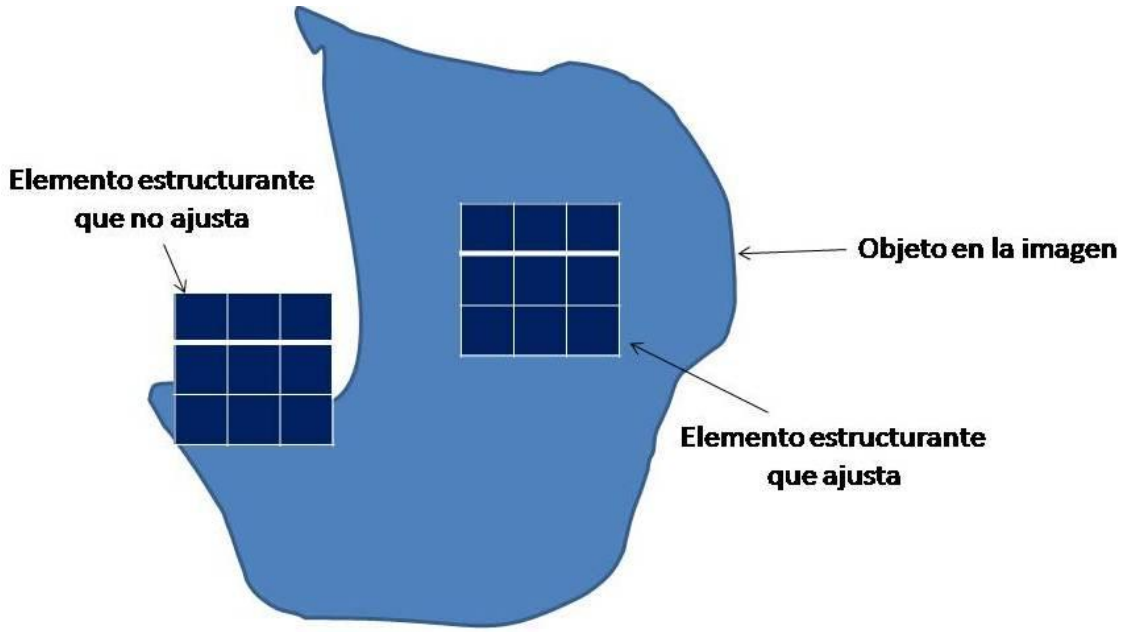

Figura 1. Operador morfológico

Hay dos operaciones típicas que son *erosión y dilatación*. Es común pensar en erosión como la contracción de un objeto en una imagen y dilatación como lo opuesto, agranda el objeto en la imagen. Ambos conceptos dependen del elemento estructurado y de cómo se ajusta en el objeto. La salida de una operación de dilatación es un píxel en primer plano para todos los puntos en el elemento estructurado a tal punto que el píxel original se ajusta en un objeto imagen. La Figura 2 muestra una imagen binaria simple que es erosionada y dilatada usando un elemento estructurado de tamaño 3x3 que está formado por unos.

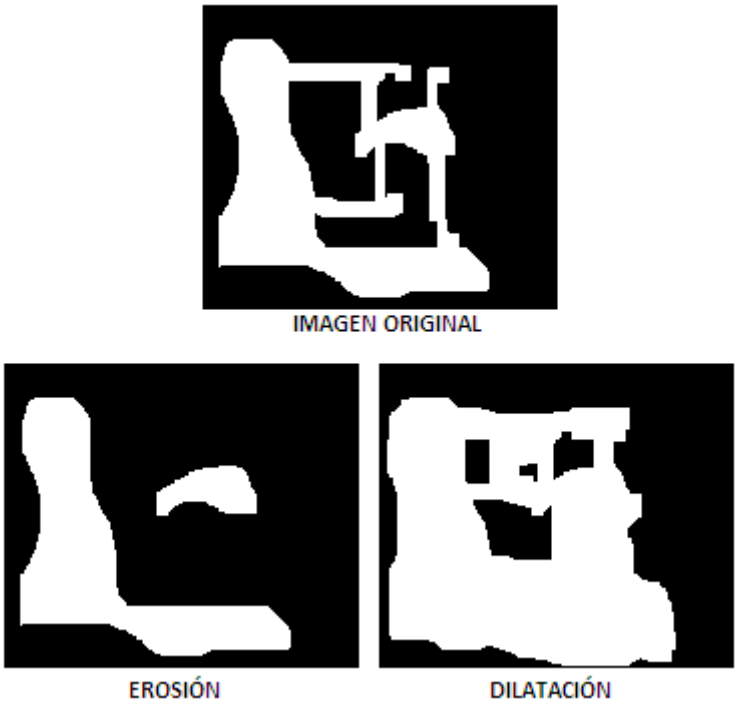

Figura 2. Efecto de erosión y dilatación

Algoritmos de orden estadístico

Los filtros de orden estadístico consisten en el procesamiento, (ordenamiento ascendente), de los píxeles de una ventana que se va trasladando a lo largo de la imagen devolviendo un píxel como salida. En estos filtros la elección del píxel de salida dependerá del orden de filtro que se desee implementar; Por ejemplo el filtro de orden 1 corresponderá al valor más pequeño de la vecindad (ventana a procesar); de la misma manera en una ventana de 9 píxeles (3x3) un filtro de orden 9 seleccionará el valor máximo. A estos dos últimos se los denomina filtros de mínimo y máximo, respectivamente. Finalmente, en una ventana de 9 píxeles, el valor intermedio (el de orden 5) corresponderá al filtro de mediana. En la Figura 3 se muestra el funcionamiento de dicho filtro, que consiste en la organización de manera ascendente de los píxeles que se encuentran dentro de la ventana, para luego obtener el valor central [5].

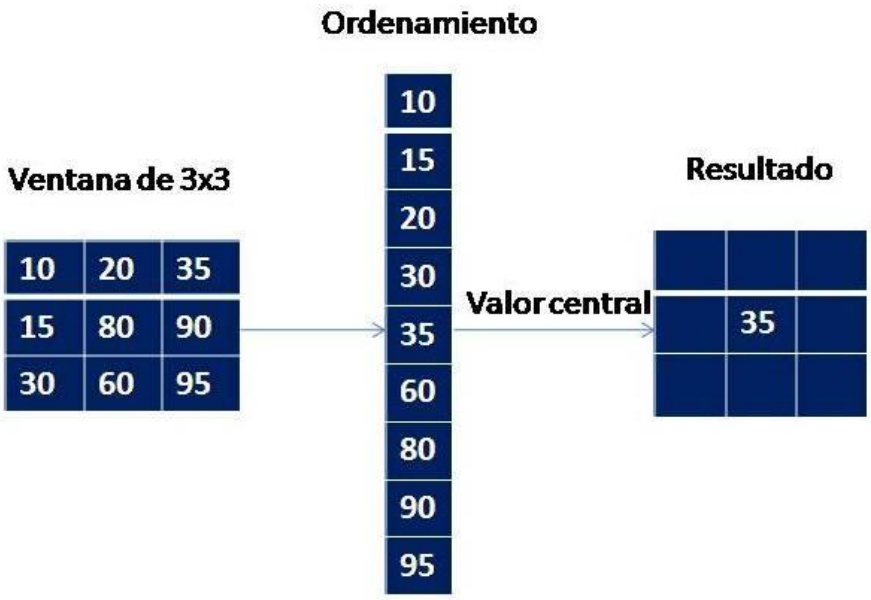

Figura 3. Filtro de mediana

El uso de este filtro permite atenuar el ruido impulsivo aleatorio (ruido sal y pimienta), mientras que los contornos de la imagen se mantienen inalterados debido a las características de los algoritmos basados en operadores de ventana.

2.1.3. Algoritmos lineales

Los métodos de análisis de imágenes lineales desarrollan las transformaciones de imagen que preservan la adición y más que la adición la noción de estructura de grupo, haciendo de la reversibilidad una característica importante [7]. Su campo de aplicación es enorme; se utilizan en geometría óptica, tomografía radiológica y cartografía entre otros. En los filtros lineales, el operador es lineal, esto quiere decir que satisface los principios de superposición y proporcionalidad. La salida por lo tanto, será la convolución de la entrada o la respuesta impulsiva del filtro.

Los algoritmos lineales se pueden clasificar en:

- La invariancia a las traslaciones:

  - Filtros lineales no invariantes a la traslación (LNSI).

  - Filtros lineales invariantes a la traslación (LSI).

- En función del número de etapas pueden ser unidimensionales, bidimensionales, ... , n-dimensionales.

- Según la dependencia de la señal de salida

  - Filtros lineales recursivos, que requieren muestras de la señal de entrada junto a muestras evaluadas de la señal de salida, de la iteración anterior, para evaluar la señal de salida.

- Filtros lineales no recursivos, que requieren solo de muestras de la señal de entrada para evaluar la señal de salida.

Los filtros digitales lineales bidimensionales son muy utilizados en el procesamiento digital de imágenes, tal como se mencionó previamente. Los más importantes son:

- Filtros pasa bajos, que reducen un suavizado de la imagen y reducen los componentes espaciales de alta frecuencia del ruido.

- Filtros pasa altos, que realzan las características de bajo contraste cuando son superpuestos sobre un fondo muy oscuro o muy claro.

- Filtros pasa banda, que son de gran aplicación ya que permiten agudizar los bordes y realzar pequeños detalles.

Algoritmos lineales basados en convolución

Cuando se convoluciona una imagen con una máscara se está aplicando un filtro cuyo tipo está determinado por el contenido de la máscara. No es muy intuitivo imaginar qué tipo de filtrado se va a obtener partiendo de una determinada máscara, salvo en las máscaras usadas para el filtrado de media, en la detección de puntos y en el de líneas horizontales y verticales

Entre los algoritmos lineales basados en convolución, se desarrollará el algoritmo de *Sobel* que es ampliamente utilizado en el procesamiento de imágenes, particularmente dentro de los algoritmos de detección de bordes. El detector de bordes sobel realiza una medición del gradiente en las coordenadas x e y con diferentes máscaras de filtro para cada dimensión. Además, del operador de sobel existen otros algoritmos lineales cuyo procesamiento es similar, tales como Robert y Prewitt, en pocas palabras solo se deben cambiar las máscaras a utilizar en la convolución. En este tipo de algoritmos las regiones con gran cambio en el gradiente corresponden a los bordes.

La convolución se realiza entre dos matrices, la matriz imagen y la máscara de convolución, obteniéndose un filtrado de acuerdo con esta última. Para obtener la convolución se realiza el producto ponderado de la matriz de convolución con el entorno de un píxel (ventana), de manera repetitiva para cada pixel de la imagen.

Típicamente, el núcleo del filtro sobel consta de un par de matrices de 3x3 y está diseñado de tal manera que se pueda tener una respuesta máxima de los bordes que van en direcciones verticales y horizontales a través de la imagen. A continuación se muestra la máscara C_V para resaltar las líneas verticales y la C_H para las líneas horizontales.

$$Cv = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} C_H \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

De este modo, el cálculo del gradiente consiste en aplicar la convolución con cada una de las máscaras anteriores, sumando luego sus resultados. El resultado C estará limitado a un valor que va de 0 a 255 cuando se trabaja sobre imágenes de 8 bits.

$$C = |Cv| + |C_H|$$

2.1.4. Procesamiento de los algoritmos de tipo espacial

Para el procesamiento eficiente de algoritmos de tipo espacial se propone dividir el tratamiento en etapas teniendo en cuenta las características comunes a todos estos. Por un lado, se propone realizar la etapa de recepción y almacenamiento de los datos en memoria compartida, luego la organización de los píxeles de a tres filas para generar las ventanas necesarias para facilitar el procesamiento y por último en función de las características paralelizables de los datos, realizar la ejecución de dichos algoritmos utilizando dos procesadores (ver Figura 4).

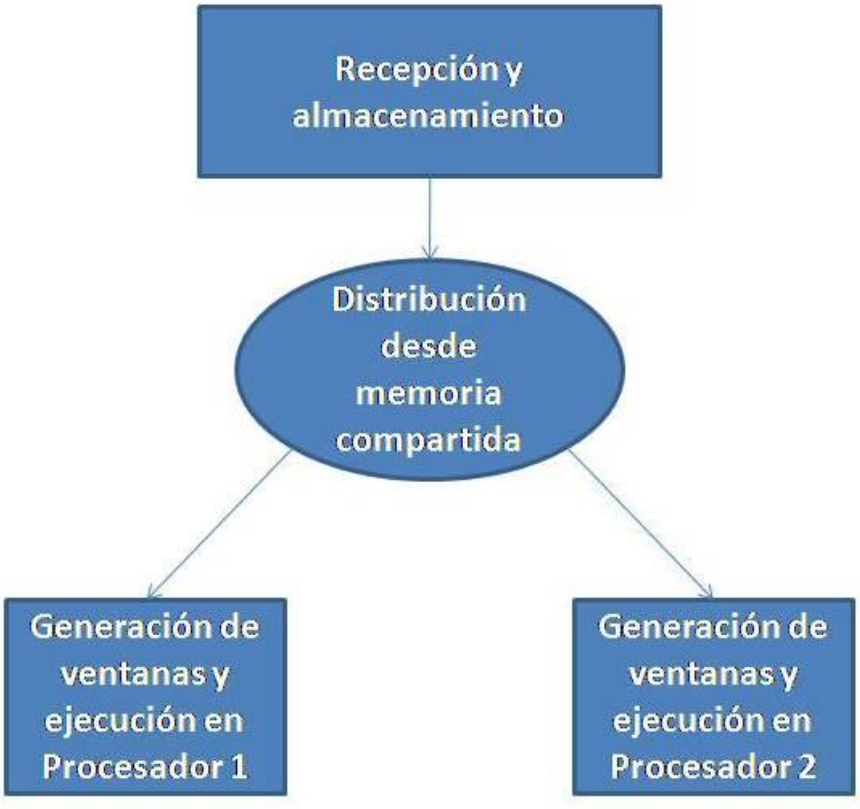

Figura 4. Etapas de procesamiento

Etapa de generación de ventana

Esta etapa deberá proveer los píxeles necesarios como para disponer de una ventana válida al sistema de procesamiento. Para conseguir esto, se pueden implementar arreglos suficientemente grandes como para contener tres filas de la imagen desde donde cada procesador extraerá las ventanas de 3x3. En la Figura 5 se muestra como debería almacenarse la imagen en los arreglos para la obtención de las ventanas necesarias, donde para una imagen de 128 x 128 píxeles el buffer debería poder almacenar 384 píxeles, de los que se obtiene un conjunto de ventanas válidas.

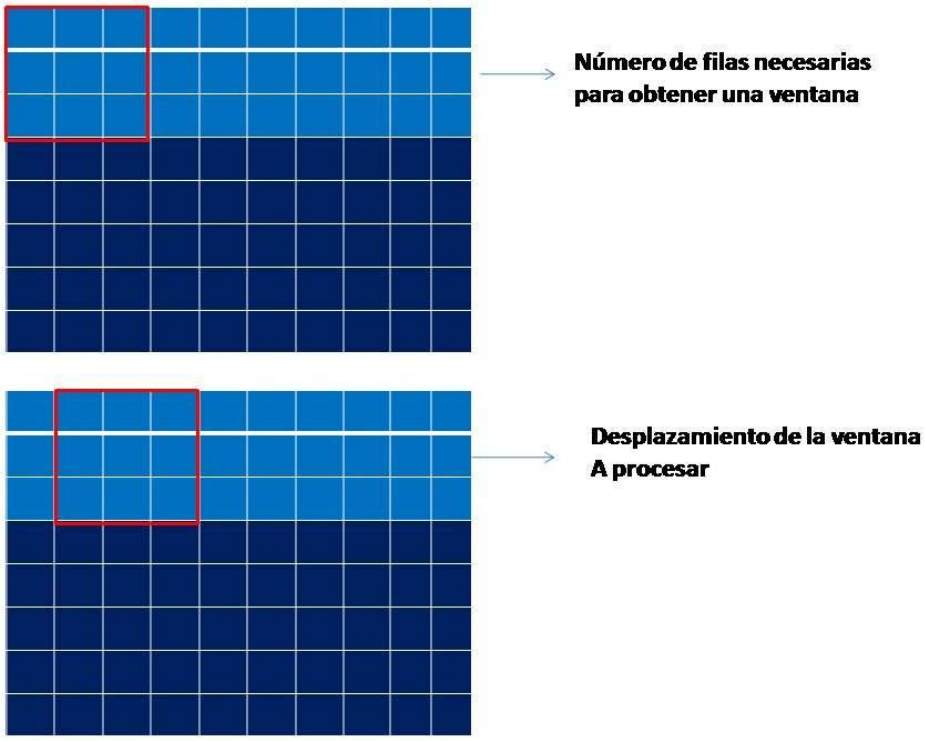

Figura 5. Generación de ventana mediante búfer FIFO

Etapa de procesamiento

Luego de obtener una ventana válida, cada procesador podrá comenzar con la ejecución del algoritmo [8]. Aquí se puede comenzar a ejecutar el algoritmo una vez que se tienen las tres filas de píxeles necesarias, para esto se debe tener en cuenta que cada vez que se lea una nueva fila se dispondrá de un nuevo conjunto de ventanas válidas.

En la etapa de procesamiento se ejecuta la tarea específica que requiere cada algoritmo, que por lo general son comparaciones, operaciones aritméticas, operaciones lógicas y asignaciones. Esta etapa se puede ejecutar completamente en cada procesador o se puede realizar mediante un coprocesador para acelerar el cálculo. En la sección siguiente se describen las características de los diferentes sistemas multiprocesador.

2.2. Arquitecturas Multiprocesador en FPGAs

Es cada vez más común el uso de arquitecturas de múltiples procesadores en todo tipo de dispositivos, desde supercomputadoras hasta sistemas embebidos.

Los dispositivos lógicos programables son cada vez más elegibles no solo por su potencialidad, sino también por la versatilidad que proveen para personalizar los diseños en sistemas digitales. Los dispositivos FPGAs vienen provistos con más y mejores innovaciones tecnológicas que permiten potenciar aún más los diseños, desde bloques de memoria, bloques multiplicadores – desplazadores, hasta potentes microprocesadores como el PowerPC.

Una de las grandes ventajas de los dispositivos FPGAs respecto de los ASICs, es el bajo coste de desarrollo respecto de la fabricación de estos últimos. Además, las herramientas de desarrollo de las FPGAs son mucho más accesibles en cuanto al coste y proveen soporte para aplicaciones de toda índole. La gran ventaja respecto a los dispositivos procesadores, es que las FPGAs son versátiles y posibilitan el diseño de sistemas digitales que combinan circuitos lógicos y procesadores, con características reconfigurables.

Las FPGAs actuales son suficientemente grandes como para contener grandes diseños, esto es, sistemas digitales completos con procesadores y todo el HW que requiere la aplicación. Estos sistemas son denominados SOPC (system on prog chip, circuito integrado por diversos módulos vlsi). Las comunidades de desarrollo y los fabricantes de FPGAs proveen los denominados bloques IP (intelectual properties), que son bloques funcionales reusables que facilitan el diseño y proveen multiplicidad de funcionalidades comunes en los SOPC. Para la implementación de procesadores, se utilizan los soft-processors que son microprocesadores descritos en VHDL y fácilmente integrables en las FPGAs.

Antes de describir las características de los sistemas multiprocesador en las FPGAs, se realizará una clasificación de los mismos según Flinn [9], que propuso lo siguiente:

- SISD: Corresponde a los sistemas simple procesador, donde las siglas significan flujo simple de instrucciones y flujo simple de datos (single instruction stream, single data stream)
- SIMD: Estos sistemas ejecutan el mismo código en varios procesadores sobre distintos datos. Las siglas representan flujo simple de instrucciones y flujo múltiple de datos (single instruction stream, multiple data stream)
- MISD: Cada procesador ejecuta diferentes códigos sobre un único flujo de datos, estos sistemas son poco comunes, salvo en aplicaciones específicas. Las siglas significan Flujo múltiple de instrucciones y flujo simple de datos (multiple instruction stream, single data stream)
- MIMD: Aquí cada procesador ejecuta su propio código sobre múltiples flujos de datos. MIMD representa flujo múltiple de instrucciones y flujo múltiple de datos (multiple instruction stream, multiple data stream)

En este punto se debe aclarar que en el sistema implementado cada procesador ejecuta la misma porción de código del algoritmo (clasificación SIMD), debido a que la gran ventaja que

tienen los algoritmos de procesamiento de imágenes de tipo espacial es que ejecutan el mismo código sobre distintos grupos de datos.

2.2.1. Características de la memoria y las comunicaciones en los modelos MIMD

Dentro de los sistemas MIMD, se pueden identificar dos tipos en función de la organización de la memoria y al modo en que se interconectan los procesadores:

- *Sistemas con memoria centralizada:* Es posible utilizar memoria centralizada o compartida cuando los sistemas multiprocesador están formados por pocos procesadores, en este caso se puede acceder a la memoria a través de un bus compartido (Figura 6). Usando memorias caché eficientes, la memoria y el bus compartido pueden proveer los datos a un conjunto reducido de procesadores. Estos sistemas se denominan SMP (symmetric multiprocessing) debido a que todos los procesadores acceden a los datos de la misma forma, (forma simétrica). Aquí la comunicación entre procesadores se realiza de forma implícita a través de la memoria.
- *Sistemas con memoria distribuida:* Cuando se tiene una gran cantidad de procesadores se utiliza otra forma de comunicación entre los mismos que facilite el sincronismo. Para esto, se asigna memoria de forma individual a cada procesador.

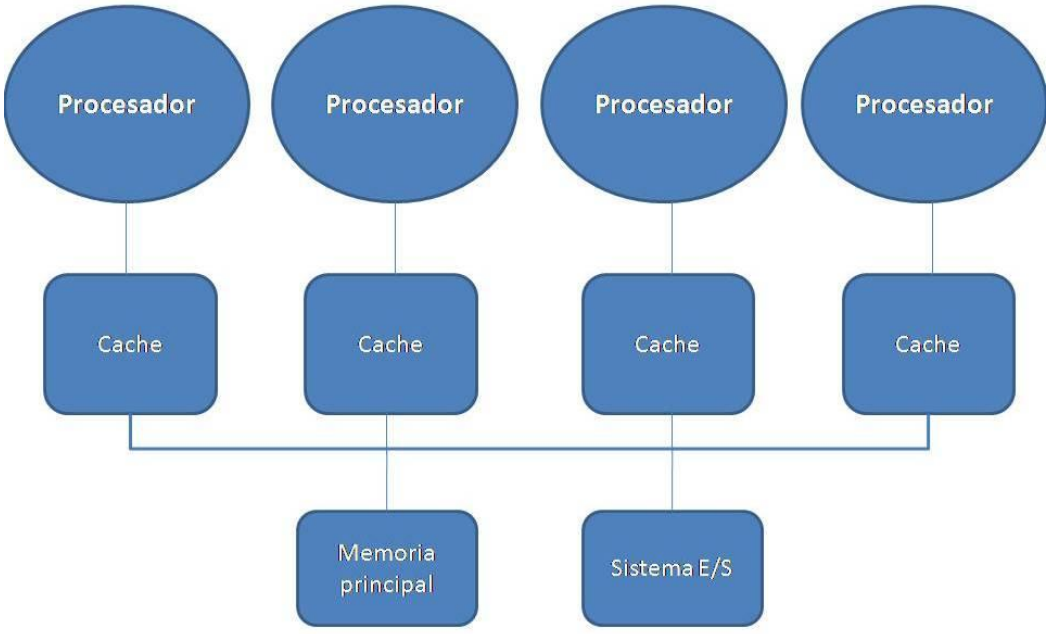

Figura 6. Sistema con memoria centralizada

Para la comunicación entre procesadores se hace necesario implementar una red de interconexión mediante un bus, tal como se muestra en la Figura 7.

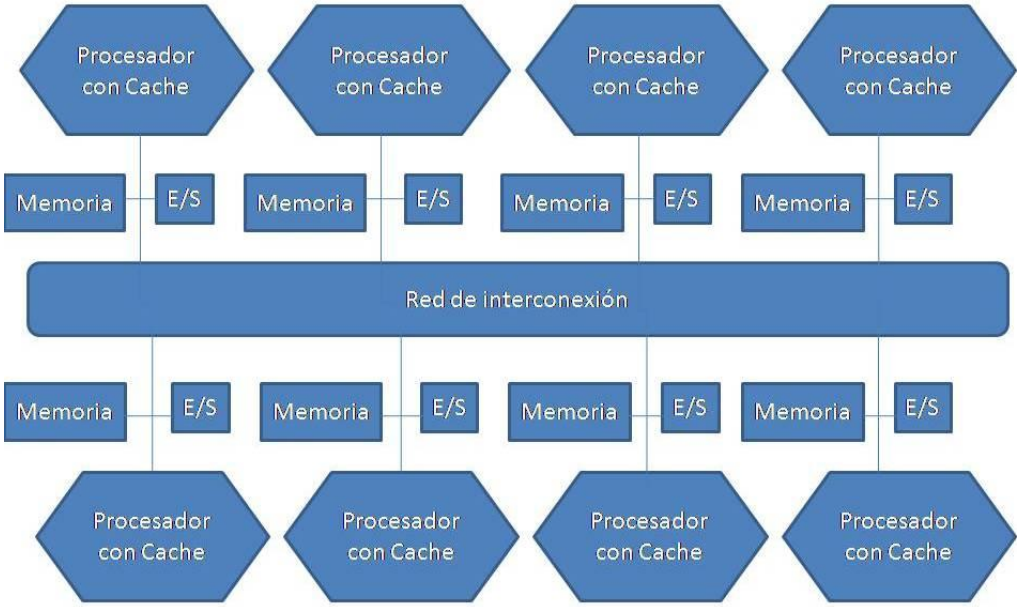

Figura 7. Sistema de memoria distribuida

Independientemente de la organización de memoria, los sistemas multiprocesador se pueden comunicar de dos maneras posibles [10-12]:

- *Variables compartidas*, donde todos los procesadores tienen acceso a toda la memoria del sistema a través de un único espacio de direcciones y el intercambio de información se realiza mediante operaciones de lectura y escritura en memoria.
- *Pasaje de mensajes*, donde cada procesador tiene su propio espacio de direcciones y los distintos procesadores intercambian información a través de mensajes.

Teniendo en cuenta las características de las tecnologías FPGAs disponibles para la implementación del sistema, se utilizará una de las siguientes clasificaciones realizadas en [13], que tienen en cuenta la organización de la memoria y el modelo de comunicación en los sistemas MIMD:

- GMSV (*Global Memory/Shared Variables*): memoria centralizada y variables compartidas.
- GMMP (*Global Memory/Message Passing*): memoria centralizada y paso de mensajes.
- DMSV (*Distributed Memory/Shared Variables*): memoria distribuida y variables compartidas.
- DMMP (*Distributed Memory/Message Passing*): memoria distribuida y paso de mensajes

Multiprocesamiento SMP o UMA

Los sistemas de multiprocesamiento simétrico (SMP - Symmetric MultiProcessing) son de especial interés por ser los implementados comúnmente en las FPGAs. En estos sistemas multiprocesador 2 o más procesadores idénticos están conectados a una misma memoria principal compartida y a una misma interfaz de entrada/salida [13]. Se dicen "simétricos" porque todos los procesadores utilizan el mismo mecanismo para acceder a la memoria y a los

periféricos y compiten en igualdad de condiciones para obtener el manejo. El nombre de UMA (Unified Memory Acces) hace referencia al acceso a memoria unificada, a diferencia de los sistemas multiprocesador con una arquitectura de memoria no uniforme que se los conoce como NUMA (Non Uniform Memory Acces) [14].

La comunicación entre los distintos procesadores en los sistemas SMP se hace por medio de la memoria compartida, por eso se los puede encasillar dentro de la categoría MIMD-GMSV. Estos sistemas se pueden implementar con varios procesadores de propósito general o específico conectados a una misma placa con soporte para los buses y la memoria.

3. Métodos algebraicos en el procesamiento de imágenes

En procesamiento de imágenes hay varios algoritmos que se pueden encasillar en la categoría operadores de ventana. Los operadores de ventaneo usan una ventana, conjunto de píxeles, para calcular su salida. El píxel que se encuentra en el centro de la ventana se denomina original y es el resultado del procesamiento.

3.1. Filtros de ordenamiento por rango

Este es un tipo de algoritmo común en el procesamiento de imágenes, en donde tienen la característica de ser no lineales y permiten remover el ruido y el alisado [6]. El filtro de mediana que es un filtro de ordenamiento por rango, es muy utilizado en la eliminación de ruido "sal y pimienta". En el filtro de mediana se toman los 9 píxeles de la ventana de 3x3 y luego de ordenarlos por rango, el que se encuentre en la posición 5 será el valor de salida.

Con este mismo método se pueden implementar los filtros de máxima y de mínima, que consisten en tomar el píxel de la posición 9 para el primer caso y el de la posición 1 para el segundo. Usando el filtro de máximo o de mínimo se logrará respectivamente una dilatación o erosión de la imagen, la razón de esto es que el resultado de un filtro de ordenamiento por rango de orden mínimo es el valor mínimo en la zona del píxel, que es exactamente lo que una operación de erosión hace. Esto también se aplica para un filtro de ordenamiento por rango de orden máximo y la operación de dilatación. Se debe destacar que estas dos operaciones son consideradas parte de los operadores morfológicos.

3.1.1. Procesamiento morfológico erosión y dilatación

La erosión y dilatación pueden ser representadas matemáticamente mediante las siguientes fórmulas:

Erosión*:* $A \ominus B = \{x: B + x < A\}$

Dilatación: $A \oplus B = U\{A + b: b \in B\}$

, donde A es la imagen de entrada y B es el elemento estructurado (ventana).

La morfología en escala de grises es más poderosa y más difícil de entender, el concepto es el mismo pero en lugar de ajustar el elemento estructurado en un objeto de dos dimensiones, se hace a través de ya sea un ajuste o no ajuste de un objeto tridimensional.

Los elementos estructurados binarios son elementos estructurados que han sido aplanados en la morfología de escala de grises. La combinación de imágenes en escala de grises y elementos estructurados en escala de grises puede ser poderosa. Una de las mejores características del procesamiento de imágenes morfológico se extiende desde el hecho de que los operadores básicos aplicados en diferentes órdenes, pueden producir muchos diferentes

resultados útiles. Por ejemplo, si la salida de una operación de *erosión* se *dilata*, la operación resultante se denomina *apertura*. El proceso inverso a la apertura, es llamado *clausura* y es una *dilatación* seguida de una *erosión*. Estas dos operaciones morfológicas secundarias pueden ser usadas para restauración de imágenes, y su uso iterativo puede producir nuevos resultados interesantes, como esqueletización y granulometría de una imagen de entrada. La Figura siguiente muestra un ejemplo de una operación binaria *apertura* y *clausura*, donde se usa un elemento estructurado de 3x3 formado por todos unos.

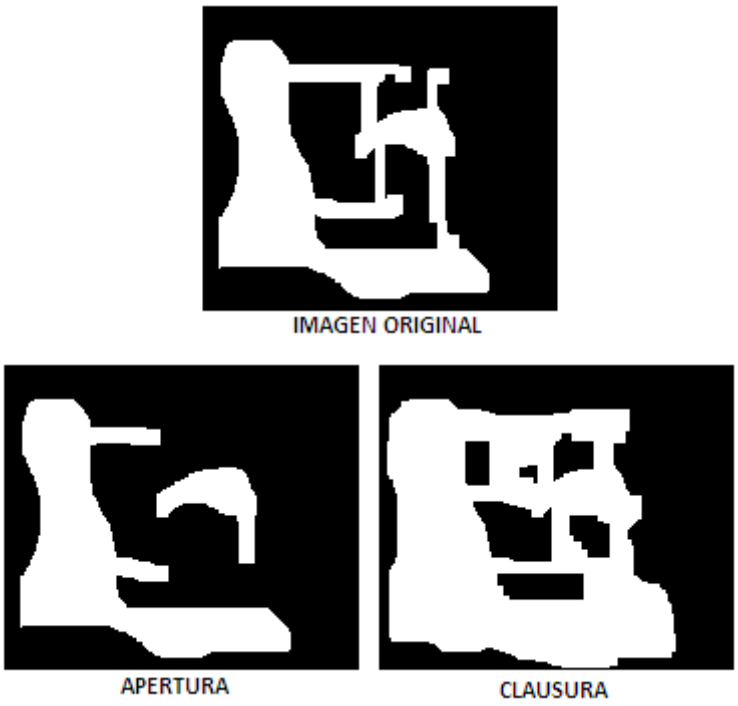

Figura 8. Apertura y Clausura de una imagen

Como se mencionó anteriormente la operación de erosión y dilatación se puede implementar mediante el filtro de ordenamiento por rango, de cualquier manera, el filtro de ordenamiento por rango solo trabaja como una operación morfológica con un elemento estructurado plano. Esto es porque la ventana del filtro de ordenamiento por rango trabaja como una clase de elemento estructurado formada por todos unos. Aun así, esta es una característica poderosa, ya que la morfología en escala de grises usando elementos estructurados planos describe los usos más comunes de la morfología. En una imagen en escala de grises, la erosión tiende a agrandar los espacios oscuros (reducir los espacios claros), y la dilatación tiende a agrandar las áreas iluminadas. Cada apertura y clausura tiende a enfatizar ciertas características en la imagen. Iterativamente, las operaciones morfológicas pueden ser usadas para elegir características específicas en una imagen, tales como las líneas horizontales y verticales [4].

3.2. Operaciones en filtros lineales

De la descripción realizada en el capítulo 2, la operación de interés en los filtros lineales de tipo espacial es la convolución.

3.2.1. Operación de Convolución

La convolución es comúnmente usada en algoritmos de sistemas de procesamiento de señales digitales. Los filtros espaciales utilizan un conjunto de máscaras (kernels) para obtener diferentes resultados en función de la operación deseada. Por ejemplo, algunas máscaras implementan el alisado, el filtrado pasabajos o la detección de bordes. La convolución, modifica la apariencia de cada píxel en función del valor de luminancia que toman los píxeles de su entorno [7].

Es sabido que los filtros de convolución pueden caracterizarse por su respuesta al impulso unitario, llamado δ(m) en el caso unidimensional y δ(m,n) en el bidimensional. En este último caso, que es el que se aplica a las imágenes, la respuesta a un delta es la salida que presenta el sistema cuando a la entrada se aplica un punto de luminancia infinita y ancho nulo. Así, la operación de convolución se define matemáticamente, para el caso bidimensional continuo como:

$$g(m,n) = h(x,y) \otimes f(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} h(x-x',y-y')f(x',y')dx'dy'$$

En el caso discreto, las integrales se transforman en sumatorias como se muestra a continuación:

$$g(m,n) = \sum_{m'=-\infty}^{\infty}\sum_{n'=-\infty}^{\infty} h(m-m',n-n')f(m',n')$$

La convolución se realiza entre dos matrices, la matriz imagen f(m,n) y la máscara de convolución h(m,n), obteniéndose un filtrado de acuerdo con esta última. Cuando se convoluciona una imagen con una máscara, se está aplicando un filtro, cuyo tipo viene determinado por las características de la misma, tal como se muestra en la Figura 9.

Figura 9. De izquierda a derecha: máscaras para filtrado de media, detección de puntos, líneas horizontales y verticales

Para obtener la convolución, se realiza el producto ponderado de la matriz de convolución con el entorno del píxel, para cada pixel de la imagen. En otras palabras, por cada ventana de

píxel de entrada, los valores de la ventana son multiplicados por la máscara de convolución. Luego, los resultados son sumados y dicha suma se divide por el número de píxeles de la ventana. Este valor es el píxel de salida en la ubicación del "píxel original", (píxel central de la ventana), en la imagen de salida.

La ventana del pixel de entrada es siempre del mismo tamaño que la máscara de convolución. El píxel de salida es redondeado al entero más cercano. La Figura 10 muestra la ventana de píxeles de entrada, la máscara de convolución, y el resultado de la salida [4]. La máscara de convolución en este ejemplo es usada como un filtro para eliminar ruido y el resultado de la operación se obtiene de la siguiente forma:

Salida de la convolución = (50*1+10*1+20*1+ 30*1+70*2+90*1+40*1+60*1+80*1)/9 = 57.77≈ 58

Un aspecto importante del algoritmo de convolución es que soporta una gran variedad de máscaras, cada una con su propia característica. La flexibilidad que tiene este algoritmo permite darle una múltiple cantidad de usos; como por ejemplo para el suavizado, enfoque, desenfoque, realce de bordes, detección de bordes, etc.

Figura 10. Ejemplo del algoritmo de la convolución

Frecuentemente una imagen que ha sido convolucionada tendrá una menor variación del valor de píxel que la imagen de entrada. Por ejemplo, en la figura anterior es obvio que la operación de convolución usando el kernel indicado da como resultado una imagen más oscura.

Gradiente de una Imagen

El gradiente es el método más empleado para calcular la derivada en una imagen [7]. Para una función f(x,y), el gradiente de la misma en un punto de coordenadas (x,y) se define como el vector:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

En la práctica, existen varias aproximaciones para el cálculo del gradiente de imágenes, todas ellas basadas en el módulo del vector gradiente:

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

El método implementado es el que utilizan las denominadas máscaras de sobel, que tienen la ventaja de proporcionar una derivación y un efecto de suavizado, lo que permite atenuar el ruido. Las derivadas parciales basadas en las máscaras de sobel son:

$$G_x = (Z_7+2Z_8+Z_9)-(Z_1+2Z_2+Z_3)$$
$$G_y = (Z_3+2Z_6+Z_9)-(Z_1+2Z_4+Z_7)$$

Luego, se suman ambos valores para obtener una aproximación del gradiente. Los Zi son los niveles de gris de los píxeles solapados por las máscaras en cualquier posición de la imagen. Las máscaras que se utilizan para el cálculo de los operadores anteriores (operadores de sobel), son las de la Figura 11.

Máscaras de operadores de sobel

-1	-2	-1
0	0	0
1	2	1

-1	0	1
-2	0	2
-1	0	1

Z1	Z2	Z3
Z4	Z5	Z6
Z7	Z8	Z9

Figura 11. Máscaras para los operadores de sobel

De este modo, el cálculo del gradiente se reduce a aplicar la convolución con cada una de las máscaras anteriores, sumando luego sus resultados.

$$G = |Gx| + |Gy|$$

4. Paralelismo y sistemas multiprocesador en dispositivos FPGA

4.1. Procesador MicroBlaze

El procesador embebido MicroBlaze, es una máquina de repertorio reducido de instrucciones (RISC por sus siglas en inglés) de 32-bits optimizado para las FPGAs de Xilinx ([15] y [16]). MicroBlaze utiliza una arquitectura de Memoria tipo Harvard, es decir los accesos a la memoria de instrucciones y a la memoria de datos se hacen por separado. Se presenta en forma de "SoftCore", lo que permite que el procesador sea altamente configurable, posibilitando agregar o quitar funcionalidades de acuerdo a los requerimientos del diseño.

Las características de este procesador incluyen:

- 32 Registros de Propósito General de 32-bits cada uno.
- Palabras de instrucción de 32-bits con 3 operandos y dos modos de direccionamiento.
- Bus de Direcciones de 32-bits.
- Pipeline (Segmentación) de emisión simple (Con cantidad de etapas configurable).

Además de estas características, el procesador MicroBlaze puede ser parametrizado, para poder habilitar funciones adicionales.

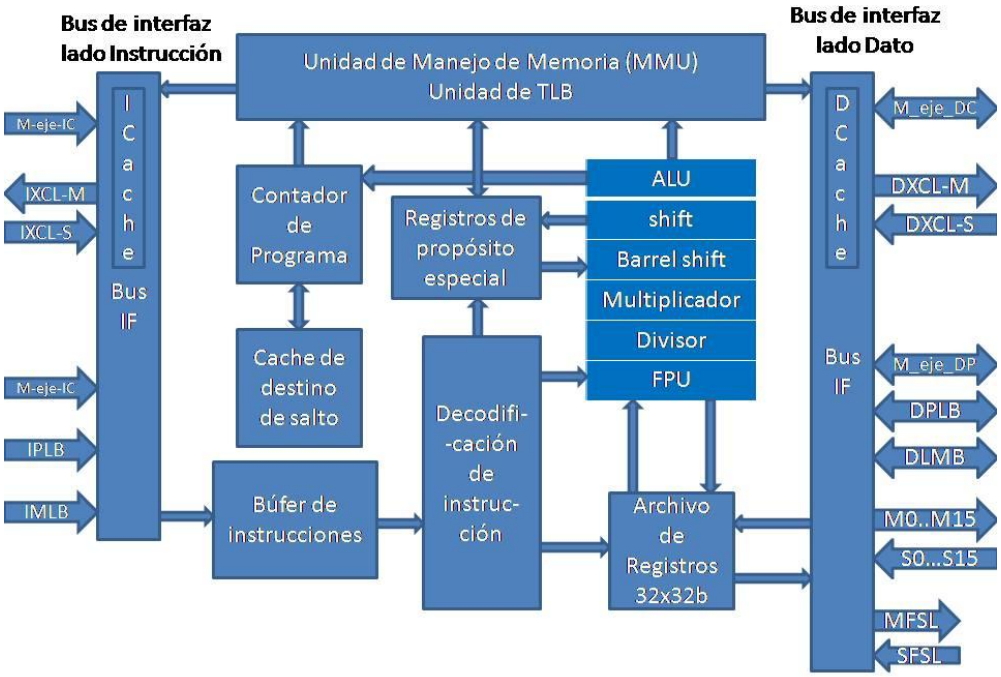

Figura 12. Diagrama en bloques del procesador microblaze

4.1.1. Buses de comunicación

Durante la implementación de un microprocesador microblaze se pueden implementar una amplia variedad de buses que posibilitan, tanto la comunicación interna del procesador como las interfaces de comunicación con otros procesadores, memorias, módulos IP y coprocesadores. En este trabajo se tuvieron en cuenta los buses LMB, PLB y FSL [16].

4.1.1.1. Bus LMB

Para conectarse con la memoria, el MicroBlaze utiliza LMB (Local Memory Bus, Bus Local de Memoria). LMB es un bus síncrono utilizado principalmente para acceder a los Block RAM de la FPGA. Usa un número mínimo de señales de control y un protocolo simple para asegurar que los accesos a esta memoria se hagan en un solo ciclo de Clock.

4.1.1.2. Bus PLB

Todo procesador necesita un bus de acceso para conectarse con los Periféricos. En el caso del MicroBlaze, se utiliza el bus PLB (Processor Local Bus, Bus Local del Procesador) [16].

PLB es un bus de alto desempeño de hasta 128-bits de Datos y 32-bits de Direcciones del tipo Maestro-Esclavo. El bus PLB permite conectar hasta 16 Maestros, con un esquema de arbitraje reparte el acceso al bus entre los distintos maestros. La prioridad de cada Maestro puede asignarse por separado, o se puede optar por un arbitraje circular (en inglés Round Robin).

Controlador de Interrupciones

Xilinx ofrece un Controlador de Interrupciones llamado "XPS Interrupt Controller", que se incorpora al bus PLB como periférico y permite concentrar hasta 32 interrupciones distintas. Este Core se debe agregar al sistema para poder manejar las interrupciones por Software.

Debugging

MicroBlaze Debug Module (MDM, Módulo de Depurado de MicroBlaze) es un core provisto por Xilinx que permite conectarse al puerto de Debugging del Procesador MicroBlaze, para poder monitorear el funcionamiento de éste en tiempo de ejecución. Este Core es indispensable para acelerar la etapa de Depurado del Software.

4.1.1.3. Bus FSL

El procesador MicroBlaze cuenta con un bus dedicado para Co-Procesadores. Este bus llamado FSL (Fast Simplex Link, Enlace Simple Rápido) utiliza FIFOs (First In First Out, Primero Entra Primero Sale) para poder comunicarse con el procesador (ver figura 13). La comunicación es unidireccional (Simplex). El bus FSL también es utilizado para comunicar dos procesadores MicroBlaze dentro de la misma FPGA [16].

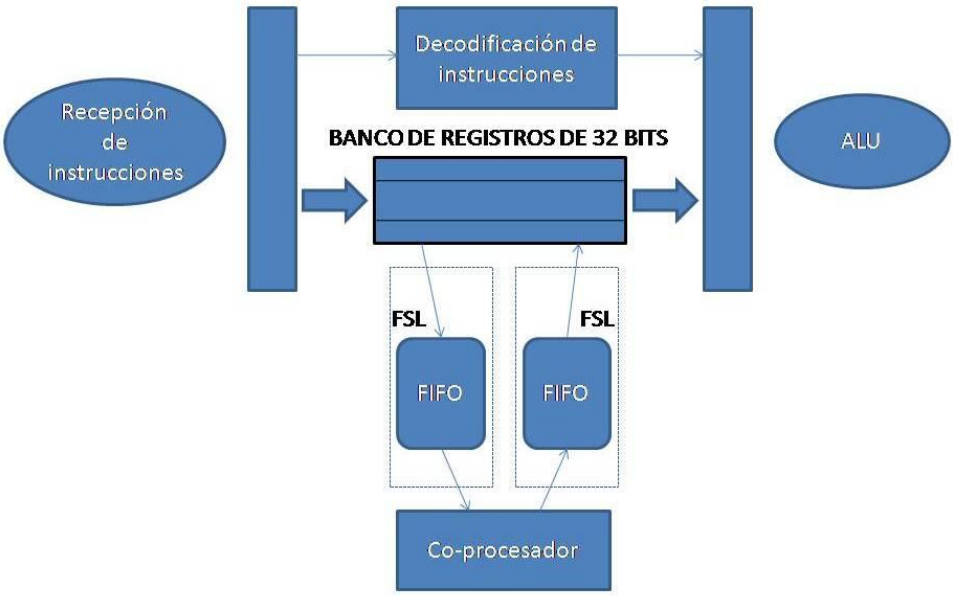

Figura 13. Estructura del bus FSL

Para obtener una interfaz bidireccional se deben tener dos buses FSL, uno para enviar datos (Master) y uno para recibir (Slave). En la Figura 14 se observa la estructura básica de dos conexiones FSL Maestro-Esclavo, con el Banco de Registros del procesador.

El bus FSL se conecta directamente con el banco de registros del procesador, lo que permite un rápido acceso a los datos, y permite al procesador seguir ejecutando otras tareas mientras el coprocesador procesa información. El ancho de palabra del bus FSL es de 32-bits.

Además, el bus FSL posee un bit extra llamado "control bit" o "bit de control", que permite señalizar la palabra que se está enviando (por el procesador o co-procesador), esto permite utilizar una palabra como encabezado de paquete o palabra de sincronismo.

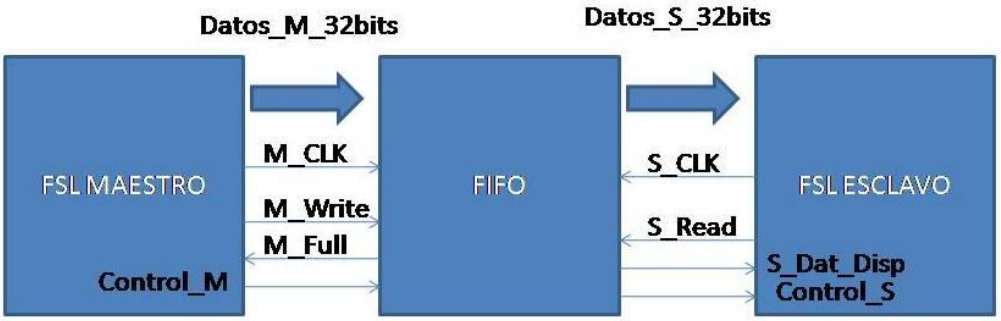

Figura 14.Esquema de conexiones en el bus FSL

- *Funcionamiento como Maestro:*

- FSL_M_Clk: Clock del lado Maestro del bus.
- FSL_M_Control: Bit de Control a enviar por el bus.
- FSL_M_Full: Bit que indica el estado de "Lleno" de la FIFO del bus.
- FSL_M_Write: Bit para iniciar una escritura en la FIFO del bus.
- FSL_M_Data: Bus de Datos FSL. 32-bits.

- *Funcionamiento como Esclavo:*

- FSL_S_Clk: Clock del lado Esclavo del bus.
- FSL_S_Control: Bit de Control Recibido.
- FSL_S_Exist: Bit que indica si existen datos en la FIFO.
- FSL_S_Read: Bit para indicar una lectura de la FIFO.
- FSL_S_Data: Bus de Datos FSL. 32-bits.

Las señales de clock para cada lado de la interfaz (Maestro y Esclavo), sirven para el caso en que la interfaz sea asincrónica, es decir cuando los Clocks Maestro y Esclavo tienen distinta frecuencia, y/o distinta fase.

Ciclos del Bus

El bus FSL permite dos tipos de ciclos, que son similares a los que se observan en los dispositivos tipo FIFO.

- *Ciclo de Escritura*

Un Ciclo de Escritura sólo puede ser realizado por el Maestro del Bus FSL. Este ciclo está controlado por la señal FSL_M_WRITE. Cuando esta señal se encuentre activa en el flanco ascendente de Clock, se copiará el contenido del bus de datos y el bit de control hacia la FIFO que compone la interfaz. Si la señal FSL_M_FULL está en "1" no se deben realizar escrituras en el Bus, ya que esto provocaría pérdida de datos (La FIFO está llena, y no puede guardar más datos). En la Figura 15 se muestra el procedimiento que sigue el Maestro del Bus para escribir cuatro datos en la FIFO.

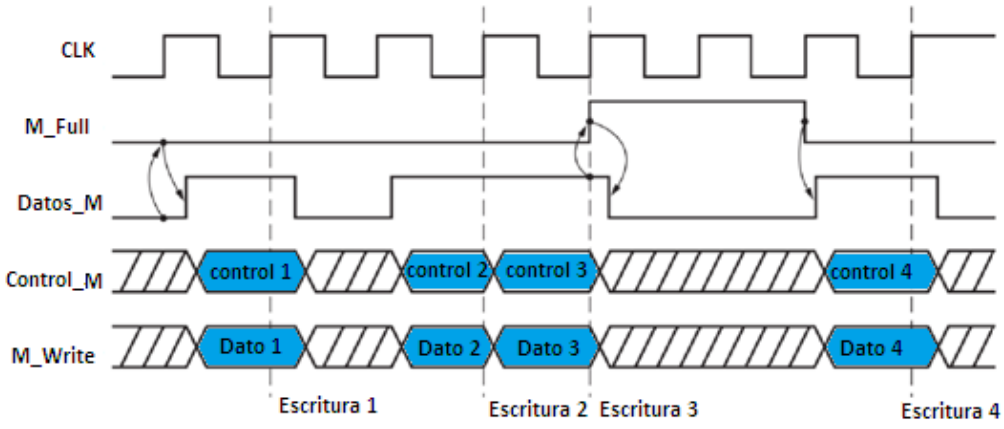

Figura 15. Ciclo de escritura

- *Ciclo de Lectura*

Este ciclo es realizado por el esclavo del bus. Cuando la señal FSL_S_EXISTS se encuentra en "1" significa que existen datos en la FIFO de la interfaz. El esclavo puede leer del bus el

primer dato que se encuentra disponible en la FIFO, luego puede indicar que leyó el dato poniendo la señal FSL_S_READ en "1". Si hay más datos disponibles en la FIFO la señal FSL_S_EXISTS vuelve a valer "1" y el esclavo debe pedir el siguiente dato poniendo la señal FSL_S_READ en "1". Luego, en el siguiente flanco ascendente de clock se copiará el nuevo dato en el bus. En la Figura siguiente se muestran tres ciclos de lectura iniciados por el esclavo sobre el bus.

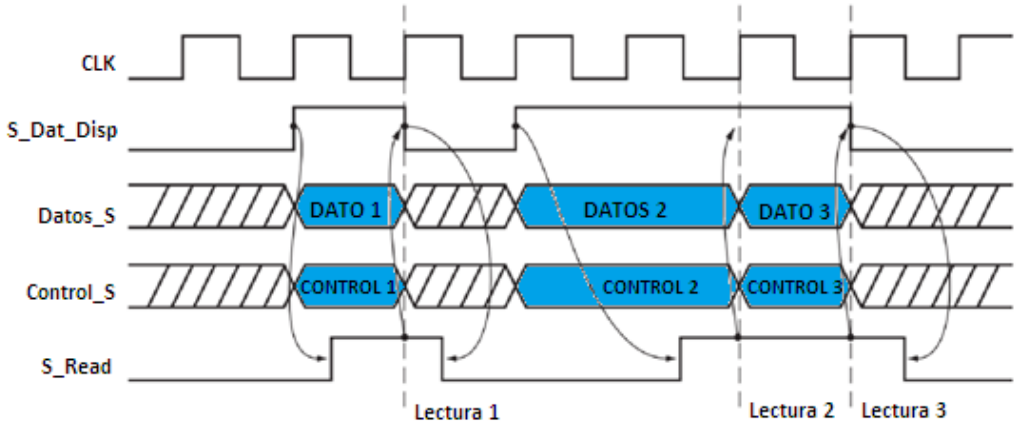

Figura 16.Ciclo de lectura del bus

4.2. Sistemas multiprocesador en FPGAs

La arquitectura multiprocesador consiste en un sistema de bus compartido con 32 bits de espacio de direcciones de memoria. Se debe destacar que el tamaño de memoria asignada y la frecuencia de reloj son totalmente configurables.

Para la implementación, cada procesador microblaze tiene memoria para almacenamiento de datos y programa. Todos los procesadores pueden acceder a una memoria compartida. Toda la lógica incluidos los microblaze y las memorias funcionan con un clock seleccionable, donde a mayor frecuencia se tiene mayor consumo. La comunicación entre procesadores se puede realizar mediante memoria compartida o pasaje de mensajes.

La herramienta para el diseño y configuración de los procesadores es el *Xilinx Dual Processor Reference Designs suite*. Los posibles diseños que permite la herramienta abarcan diferentes arquitecturas dual-core basadas en los procesadores MicroBlaze y PowerPC® [17].

4.2.1. Detalles específicos de un sistema multiprocesador en FPGA

El Dual Processor Reference Design suite [18], tiene cuatro sistemas de referencia con varias combinaciones de los sistemas dual processor. Los cuatro diseños ilustran dos sistemas de procesadores totalmente funcionales que interactúan entre sí a través de los cores para comunicación entre procesadores.

Estos cores proporcionan memoria compartida, sincronización simple y funciones de paso de mensajes para usar entre los procesadores [17]. Sus principales características son:

- XPS_Mutex: El core XPS Mutex ayuda a lograr la sincronización entre varios procesadores al acceder a recursos compartidos. El núcleo tiene un número configurable de mutexes y tiene un esquema de bloqueo para escritura. La interfaz del software es proporcionada por el controlador mutex_v1_00_a.
- XPS_Mailbox: El core XPS Mailbox ayuda al procesador a pasar mensajes simples a otro procesador mediante una FIFO. El core es adecuado para mensajes de tamaño pequeño a mediano (unos 50 bytes). Además, ofrece una línea de interrupción que indica la presencia de datos en el buzón. La interfaz del software es proporcionada por el controlador mbox_v1_00_a.

Todos los procesadores también tienen memoria compartida (externa e interna) entre ellos, lo que proporciona un esquema de intercambio de datos específico de la aplicación.

Sincronización mediante XPS_Mutex

Los núcleos de procesamiento requieren sincronismo cuando acceden a recursos compartidos como periféricos y memoria. Existen algunas técnicas que utilizan los sistemas operativos para sincronismo entre procesos como los semáforos, pero estas técnicas no son útiles en los sistemas multiprocesador porque no hay un único sistema operativo para todos los núcleos. El procesador MicroBlaze no proporciona soporte para instrucciones atómicas de lectura-modificación-escritura. Por lo tanto, el software XPS proporciona un módulo de sincronización de hardware llamado XPS_Mutex para proporcionar la capacidad de crear regiones de exclusión mutua entre varios procesadores.

El XPS_Mutex posee un conjunto de registros que permiten configurar el valor y el identificador de procesador. Para adquirir la exclusión mutua, los procesadores escriben la ID que los identifica, (definida en el software del núcleo), al registro de exclusión mutua correspondiente y el valor de cero que representa recurso ocupado. El mutex arbitra el acceso simultáneo y almacena el ID del procesador, que primero solicitó el recurso, en el registro de valor de mutex. Si el mutex ya está bloqueado, el valor de mutex permanece sin cambios. Luego, cada procesador intenta el acceso al recurso mediante la lectura del valor de exclusión mutua y la comparación con su ID de procesador.

Se debe aclarar que el procesador que adquiere el mutex tiene la libertad de liberarlo en cualquier momento al realizar una operación de escritura en el registro de mutex con su propia ID de procesador y un valor de uno.

Pasaje de mensajes mediante XPS_Mailbox

El método de Buzones se utiliza para el pasaje de mensajes entre uno o más remitentes y un receptor. El método consiste en un canal en donde el remitente envía mensajes que van quedando en cola mediante FIFO y el receptor los lee. La recepción se puede realizar de manera sincrónica o asincrónica. El método asincrónico requiere una interrupción desde el remitente hacia el receptor, en cambio en el método sincrónico el receptor continúa activamente sondeando a la espera de nuevos mensajes. Xilinx provee el XPS-Mailbox para comunicación entre procesadores, en donde cada mailbox tiene dos FIFOs, una para transmitir y una para recibir.

Cada buzón tiene un par de interfaces para conectarse a los procesadores que se comunican a través del buzón. Sin embargo, es posible conectar varios procesadores a cada interfaz, el uso recomendado es usar un solo buzón entre un par de procesadores.

El núcleo de hardware del XPS_Mailbox de Xilinx generalmente es adecuado para mensajes de tamaño pequeño a mediano, generalmente de unos 50 bytes como máximo. El procesador del remitente debe copiar el mensaje completo de las memorias locales o externas y escribirlo en la FIFO. Por lo tanto, no es adecuado para mensajes grandes, ya que la utilización del procesador en esta copia de mensajes desperdicia ciclos de procesamiento.

La llegada de un mensaje en la FIFO del buzón se indica al receptor mediante el envío de una interrupción en la línea IRQ que sale del buzón. La generación de interrupciones es opcional y se puede desactivar, lo que significa que la comunicación entre el remitente y el receptor puede ser síncrona o asíncrona.

Controlador de memoria compartida

La memoria compartida es la forma más común de pasar información entre procesadores y las características principales son:

- Cualquier procesador puede acceder a cualquier ubicación de memoria directamente
- Las comunicaciones se producen mediante la lectura y escritura de la memoria
- Los datos pueden extraerse mediante múltiples procesadores por medio de una API que realiza el trabajo de lectura/escritura
- El acceso al sector de memoria compartida debe ser sincronizado por algún protocolo de HW/SW entre los procesadores

Memoria compartida es un método rápido de comunicación asincrónica que es muy eficiente para paquetes de datos de gran tamaño. En la FPGA se implementa usando el controlador de memoria MPMC, en donde cada puerto del controlador se asigna al mismo rango de direcciones, esto permite a todos los procesadores acceder a la misma memoria. Este método requiere que el programador defina las regiones compartidas y diseñe el protocolo de software para pasar información entre los procesadores. El método de memoria compartida se puede implementar con bloques de memoria local BRAM, en donde el controlador puede conectar solo dos procesadores (posee dos puertos). Adicionalmente, se puede implementar el método con memoria externa DDR2-SDRAM, en donde el controlador posee mayor cantidad de puertos de conexión.

Comunicación y sincronismo

La comunicación de memoria compartida es la forma más común y obvia de pasar información entre los procesadores. Al tener una variable global compartida o una estructura de datos en la memoria, el software de un procesador puede actualizar fácilmente el valor de la variable y hacerla visible para otros procesadores. Para esto, se debe conocer la dirección de la variable o un puntero a la región compartida. La región de código en la que se modifican los datos compartidos se conoce como una región crítica. Para el acceso a los datos compartidos

debe existir algún tipo de método no conflictivo en el que cada procesador acceda a los datos. Para lograr esto se requiere un protocolo o la construcción de sincronismo para serializar los accesos al recurso compartido. Una forma de coordinación puede lograrse mediante la primitiva XPS_Mutex en donde el procesador bloquea antes de acceder y desbloquea cuando ya no necesita entrar en la región crítica, la metodología utilizada es la de exclusión mutua.

Otra consideración importante al usar la memoria compartida es la coherencia de caché. Si los procesadores del sistema almacenan en caché la región de la memoria compartida, el usuario debe tener en cuenta las situaciones que podrían dejar la memoria caché en un estado incoherente. Ni el MicroBlaze ni el procesador PowerPC proporcionan coherencia de caché por hardware. Cuando ambos procesadores acceden a la misma memoria física, las actualizaciones de un procesador a la memoria no son vistas directamente por el subsistema de caché del otro, es por esto que es responsabilidad del software garantizar la coherencia. Una forma sencilla de garantizar la coherencia es dedicar las regiones no almacenadas en caché de la memoria principal para propósitos de memoria compartida. Una recomendación importante en la implementación de un sistema de memoria compartida es deshabilitar la cache de cada procesador al momento de utilizarla [17].

Algunos tipos de acceso a datos compartidos no requieren ninguna sincronización ni coherencia de memoria. Por ejemplo, si el modelo de datos compartidos utiliza la forma simple de varios lectores, solo se requerirá sincronismo para el escritor, los lectores no necesitan acceder a los datos compartidos de manera mutuamente excluyente entre sí. Esto quiere decir que la memoria compartida debe modelarse de acuerdo con el paradigma que se adapta mejor a la aplicación.

Otra forma de sincronización entre procesadores se puede lograr mediante la primitiva XPS_Mailbox mediante la característica de paso de mensajes entre procesadores. La API del software para paso de mensajes está orientada a la forma de realizar las llamadas de lectura() y escritura(), por lo tanto, el software puede tratar al buzón como un archivo al que se accede en serie para enviar y recibir datos. La función de paso asíncrono de mensajes permite que el software en un procesador se ejecute sin tener que perder ciclos esperando a que los datos lleguen a un buzón, lo que aísla a los remitentes lentos de los receptores rápidos que tienen otras tareas críticas que realizar.

Un requisito común de los sistemas de multiprocesamiento es que salgan del reinicio y realicen algún tipo de paso de sincronización entre ellos antes de continuar con las funciones individuales dedicadas. Por ejemplo, en sistemas de multiprocesamiento con relaciones de maestro-esclavo, el procesador maestro se encarga de inicializar el entorno operativo para todos los esclavos, después de lo cual se inicia la ejecución de los esclavos. Este tipo de sincronización se suele formular como problemas de tipo de encuentro o barrera. En el diseño del software se debe de definir el protocolo de encuentro mediante el uso de cualquiera de los esquemas de comunicación entre procesadores descritos previamente. Por ejemplo, el maestro puede enviar un mensaje de inicio a los procesadores esclavos a través de un buzón XPS_mailbox. Una segunda posibilidad es utilizar la capacidad de generación de interrupciones del buzón para interrumpir los procesadores, lo que indica un tipo de evento de barrera.

4.2.2. Sistema multiprocesador microblaze

Este diseño ilustra una topología con dos procesadores MicroBlaze en buses PLB independientes [19]. Los procesadores utilizan un core XPS_Mutex y uno XPS_Mailbox para sincronismo y pasaje de mensajes. Ambos procesadores tienen acceso a la memoria DDR externa a través del controlador de memoria MPMC. Además de compartir la memoria externa, los dos procesadores comparten la memoria local (BRAM) a través de los controladores de memoria XPS BRAM. Cada procesador tiene un controlador de interrupción asignado para manejar varias interrupciones en el sistema. Ambos procesadores tienen una interfaz de depuración a través del core MDM. La Figura 17 muestra un diagrama en bloques con el diseño del dual processor MicroBlaze [18].

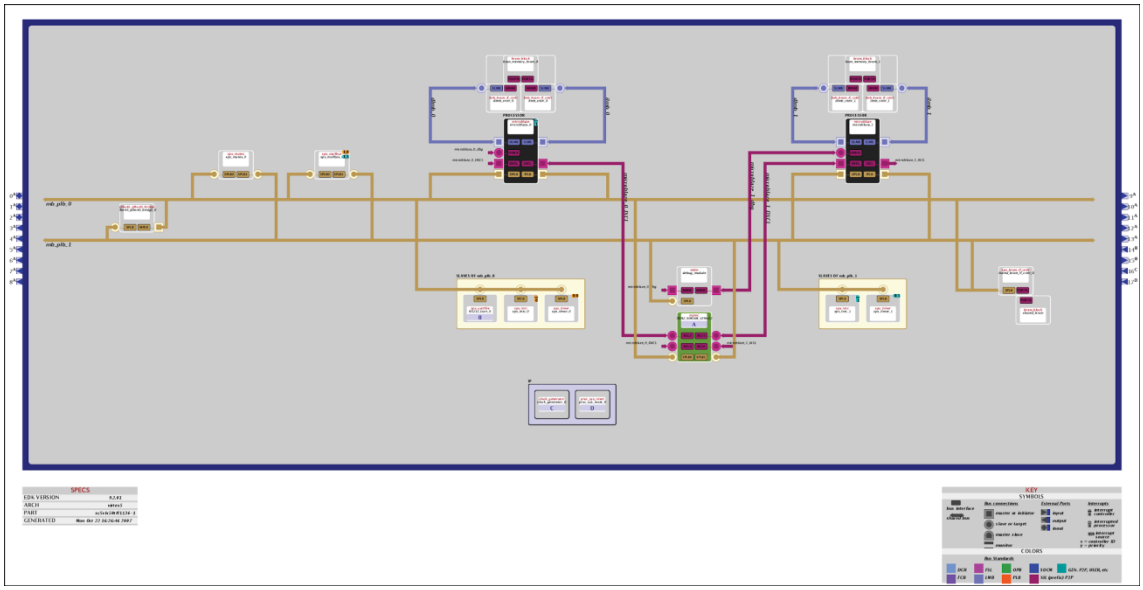

Figura 17. Diseño de un sistema de dos procesador microblaze

4.2.3. Sistema multiprocesador PowerPC

Este diseño ilustra una topología con dos procesadores PowerPC 405 en buses PLB independientes. Los procesadores utilizan un core XPS_Mutex y uno XPS_Mailbox para sincronizar y pasar mensajes. Ambos procesadores tienen acceso a la memoria DDR externa a través de la interfaz PLB. La memoria externa es almacenada en caché por ambos procesadores. Además de compartir la memoria externa, los dos procesadores comparten la memoria local (BRAM) a través de las interfaces OCM y PLB.

Hay un controlador de interrupciones asignado a cada procesador para manejar varias interrupciones en el sistema. Ambos procesadores tienen una interfaz de depuración a través del controlador jtag_ppc. La Figura 18 muestra un diagrama de bloques de este diseño.

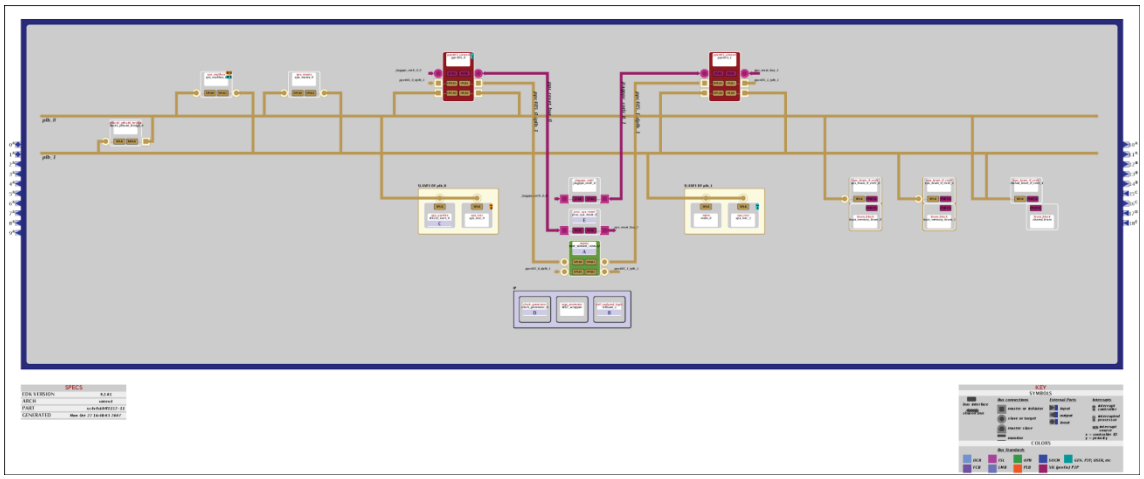

Figura 18.Sistema de dos procesadores PowerPC

4.2.4. Sistema multiprocesador PowerPC 405 y Microblaze

Este diseño ilustra una topología con un PowerPC405 y un procesador MicroBlaze, cada uno con sus propios buses PLB. Los procesadores utilizan un core XPS_Mutex y uno XPS_Mailbox para sincronizar y pasar mensajes. Ambos procesadores tienen acceso a la memoria DDR externa a través del controlador de memoria MPMC. En este diseño, además de compartir la memoria externa, los dos procesadores comparten la memoria local (BRAM) a través de los controladores de memoria XPS BRAM. Cada procesador tiene un controlador de interrupción asignado para manejar varias interrupciones en el sistema. Ambos procesadores tienen una interfaz de depuración a través del controlador jtag_ppc y los cores MDM. El core MDM también proporciona una interfaz UART para el segundo procesador.

4.3. Implementación del sistema embebido

El sistema diseñado integra dos procesadores MicroBlaze que cuenta con los buses previamente mencionados y memoria compartida en donde se almacenan los datos. Esto último, permite obtener diferentes tiempos de procesamiento de imágenes con uno y dos procesadores. Adicionalmente, se implementó una etapa de coprocesador que se encarga de aprovechar las características de concurrencia de la programación basada en la descripción de hardware (VHDL) para la ejecución de una parte del algoritmo.

Debido a la naturaleza de los datos paralelizables que tienen las imágenes, se utilizó el método de programa simple y múltiples datos (SPMD), esto significa que cada procesador puede ejecutar el mismo programa, excepto por el código que se encarga del sincronismo para la lectura y escritura de datos que se implementó mediante mailbox, (paso de mensajes), entre los procesadores. El sincronismo es necesario debido a que el Procesador MB0, encargado de recibir los datos y almacenarlos en memoria, deberá informar al procesador MB1 cuando hay datos disponibles para procesar y este último deberá informar cuando finalice el procesamiento, para devolver la imagen resultado al host.

Las imágenes a procesar fueron leídas con el software Matlab y transmitidas desde una PC (Host) por puerto serie hacia la FPGA Virtex V, (los detalles de la tecnología utilizada se muestran en el Apéndice B sección B.1), en donde se realizó todo el procesamiento y se obtuvo la imagen procesada para su posterior almacenamiento en la memoria compartida DDR2-SDRAM. Para la visualización de la imagen resultante, se utilizó un bus de datos serie UART para transmitir los píxeles obtenidos desde memoria compartida hacia el PC-Host que utiliza Matlab. En el diagrama en bloques de la figura siguiente se muestra la interconexión entre las diferentes partes del sistema.

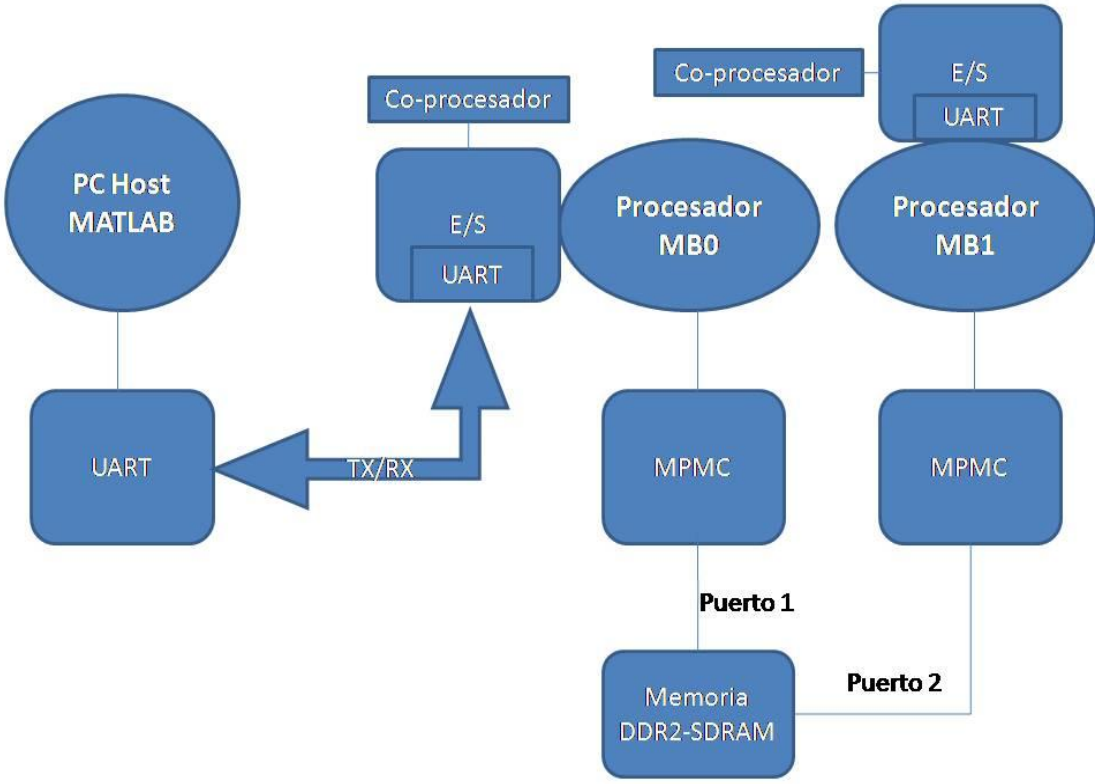

Figura 19.Diagrama en bloques del sistema completo

El sistema de procesamiento tiene en cuenta la característica de los filtros basados en regiones, donde se opera sobre una ventana móvil de píxeles. La ventana móvil típica es de 9 píxeles (ventana de 3x3) como muestra la figura 20a.

Para comprender mejor la distribución de los datos entre los procesadores, si se tiene una imagen de 8 filas, las cinco primeras son leídas por el procesador 1 (MB1), la cuarta y la quinta también por el procesador 2 (MB0) junto con las últimas tres, tal como se muestra en la figura 20b.

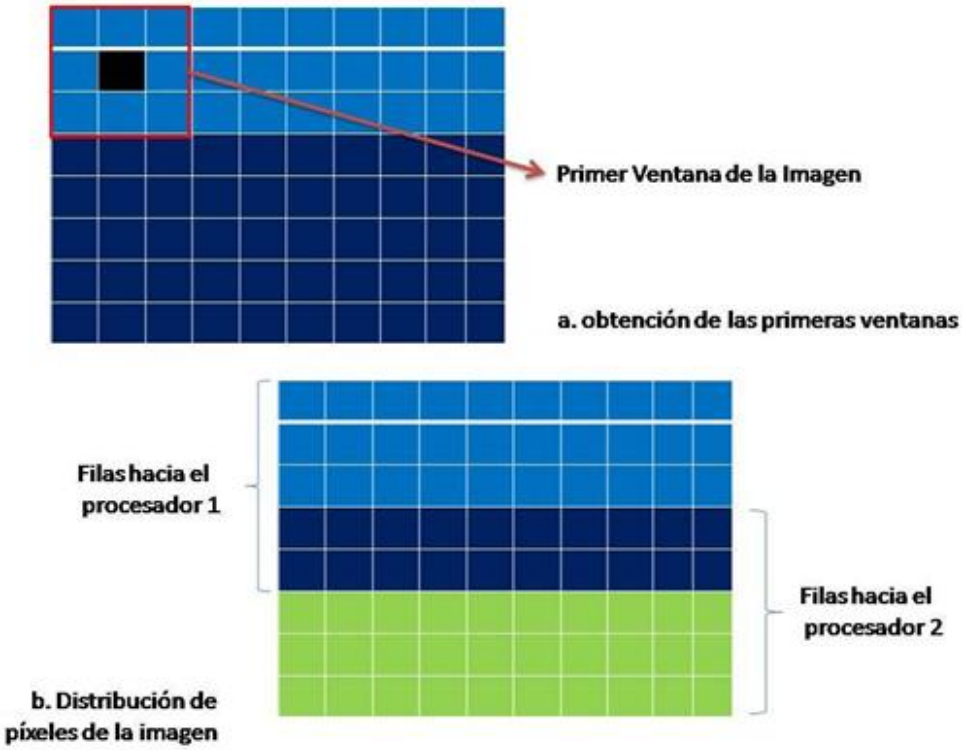

Figura 20. (a) Obtención de Ventanas. (b) distribución de filas a cada procesador

4.3.1. Etapa concurrente

Los filtros basados en operadores de ventana forman parte del procesamiento de imágenes basado en regiones. Para realizar este tipo de procesamiento es necesario seleccionar los píxeles correspondientes a la región de interés y proveerlos al coprocesador para que realice las tareas de cálculo necesarias. Como la etapa de coprocesador se implementará en lenguaje de descripción de HW ([20]y[21]), es importante conocer las características internas de una FPGA para poder determinar los recursos que han sido utilizados en el diseño, la descripción de dichas características se encuentra en el apéndice A.

A continuación se describen las características de la implementación del método de ordenamiento utilizado en los algoritmos de mediana, erosión y dilatación; como el método de convolución utilizado para el algoritmo de sobel. Respecto al algoritmo de media, debido a la simplicidad de la implementación mediante el promediado de píxeles, solo se presenta el código desarrollado en el Apéndice D sección 2.

4.3.1.1. Sistema de Ordenamiento en VHDL

La característica principal de este tipo de algoritmos es que requieren el ordenamiento de menor a mayor (o viceversa) del valor de los píxeles involucrados en la operación. Este bloque se implementó en VHDL para obtener el ordenamiento de los píxeles de la ventana de interés. Como muestra la figura 21, el bloque de ordenamiento se implementa mediante comparadores que reciben como entradas grupos de 2 píxeles para determinar cuál es mayor y cual es menor. De esta manera y mediante varios ciclos de clock se obtienen los 9 píxeles ordenados, de los

cuales se selecciona como salida el de la posición 5 para implementar la mediana y los de los extremos para implementar erosión o dilatación en imágenes en escala de grises [22].

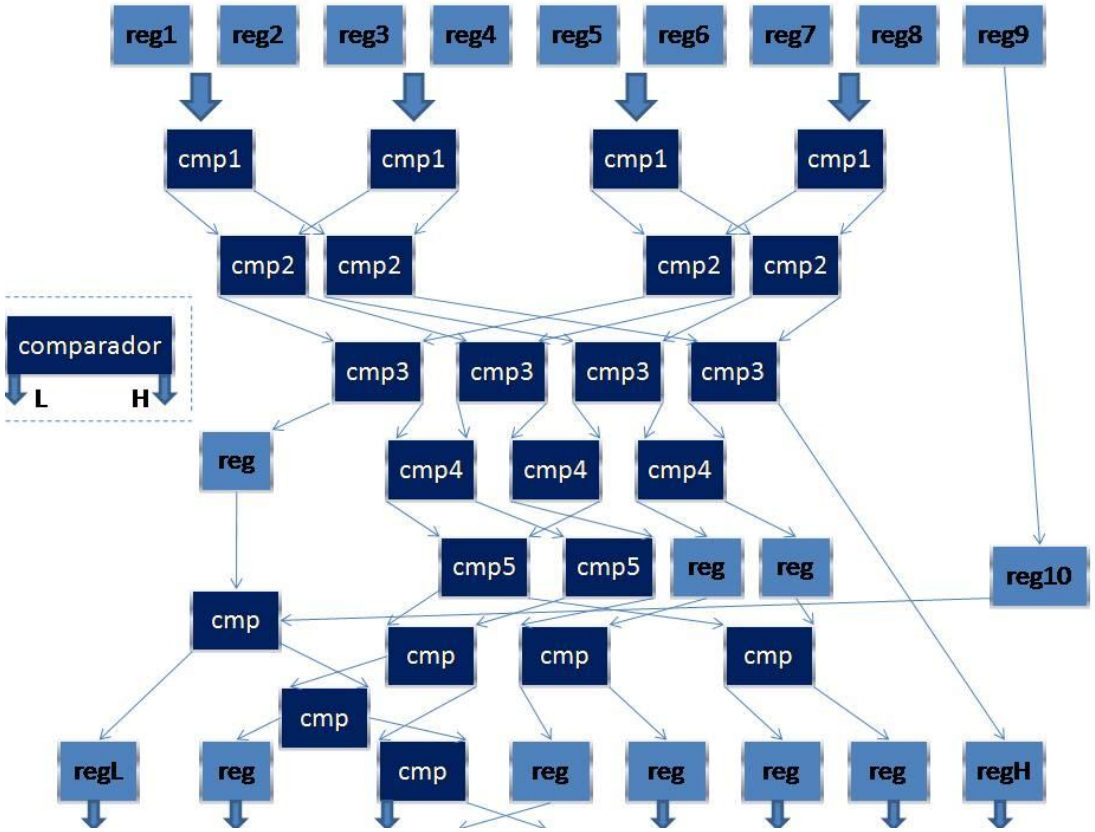

Figura 21. Diagrama en bloques del algoritmo de ordenamiento en VHDL

El código en VHDL cosiste principalmente en una sucesión de comparaciones y asignaciones que se detallan en el Apéndice D sección 1. Por ejemplo, la implementación del nivel 1 del algoritmo de ordenamiento se realiza como se muestra a continuación:

- En el nivel 1 se realiza la comparación entre los píxeles de la ventana, excepto con el N° 9 llamado v33, que no se modifica:

```
-- comparaciones nivel 1
if v11 <v12 then        -- comparación entre el pixel v11 y el v12
        c11_L <= v11;
        c11_H <= v12;
else
        c11_L <= v12;
        c11_H <= v11;
end if;
if v13 <v21 then      -- comparación entre el pixel v13 y el v21
        c12_L <= v13;
        c12_H <= v21;
else
        c12_L <= v21;
        c12_H <= v13;
end if;
```

```
if v22 <v23 then     -- comparación entre el pixel v22 y el v23
          c13_L <= v22;
          c13_H <= v23;
else
          c13_L <= v23;
          c13_H <= v22;
end if;
if v31 <v32 then                -- comparación entre el pixel v31 y el v32
          c14_L <= v31;
          c14_H <= v32;
else
          c14_L <= v32;
          c14_H <= v31;
end if;
          r11 <= v33;           --pixel v33 no se compara en esta etapa
```

4.3.1.2. Sistema de Convolución en VHDL

Para el diseño del bloque de convolución el hardware específico del código VHDL está ligado a bloques de suma y multiplicación [23].

El módulo de convolución espacial realiza el producto entre cada ventana y la máscara píxel a píxel, para luego sumar cada resultado y dividirlo por la cantidad de píxeles de la ventana. En la Figura 22 se muestra el diagrama de HW del bloque que realiza la convolución. Cabe aclarar que se realizó una división por 8 mediante desplazamientos, para evitar la utilización de excesivos recursos y la operación de redondeo.

Los detalles del código de convolución se muestran en el apéndice D sección 3, cuya implementación cosiste principalmente en:

- Realizar el producto entre los elementos de la máscara y los píxeles de la ventana

```
m0 <= signed('0'&v11)*signed(k0);
m1 <= signed('0'&v12)*signed(k1);
m2 <= signed('0'&v13)*signed(k2);
m3 <= signed('0'&v21)*signed(k3);
m4 <= signed('0'&v22)*signed(k4);
m5 <= signed('0'&v23)*signed(k5);
m6 <= signed('0'&v31)*signed(k6);
m7 <= signed('0'&v32)*signed(k7);
m8 <= signed('0'&v33)*signed(k8);
```

- Realizar las sumas correspondientes

```
a10 <= (m0(16)&m0)+m1;
a11 <= (m2(16)&m2)+m3;
a12 <= (m4(16)&m4)+m5;
a13 <= (m6(16)&m6)+m7;
a14 <= m8(16)&m8;
a20 <= (a10(17)&a10)+a11;
a21 <= (a12(17)&a12)+a13;
a22 <= a14(17)&a14;
a30 <= (a20(18)&a20)+a21;
a31 <= a22(18)&a22;
a40 <= (a30(19)&a30)+a31;
```

- Por último, realizar la división mediante desplazamientos
  d0 <= a40(20 downto 3);

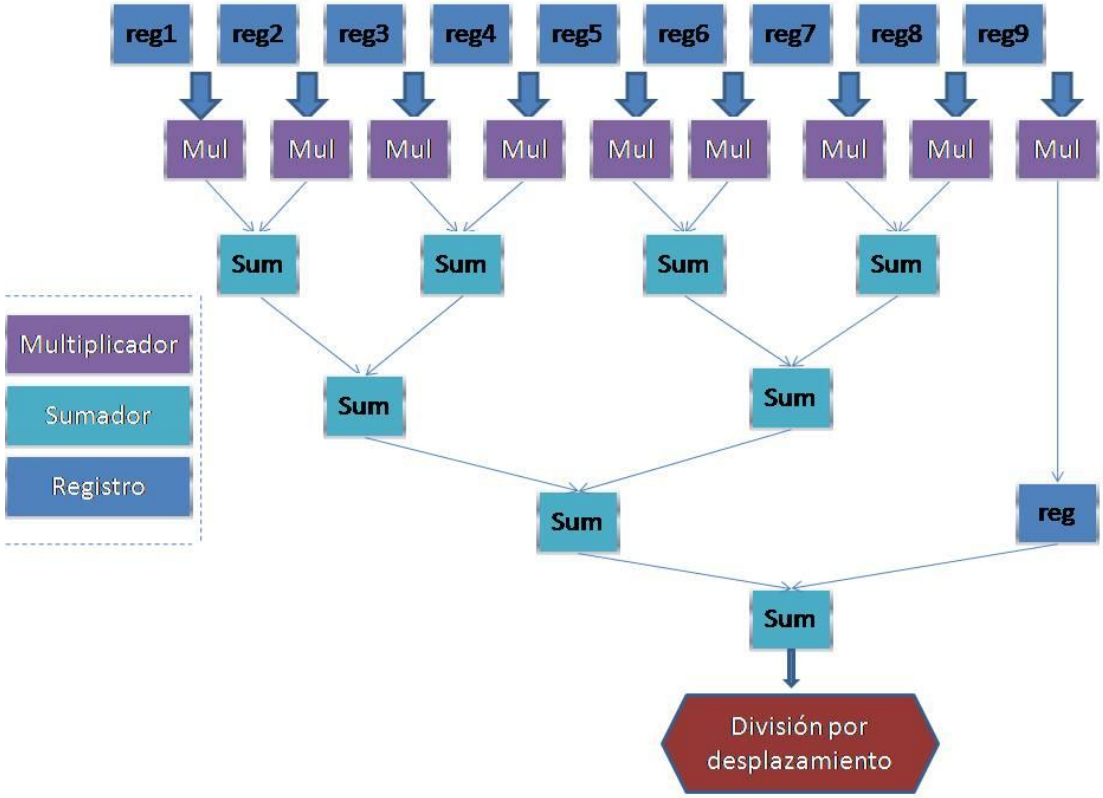

Figura 22. Diagrama en bloques de la convolución en VHDL.

4.3.2. Etapa multiprocesador

4.3.2.1. Hardware del sistema multiprocesador

Parámetros de implementación del sistema

Al crear un proyecto en XPS, el software permite mediante un "Wizard" seleccionar la FPGA (virtex 5 xc5vlx50T), el número de procesadores (ver Figura 23) y configurar automáticamente los parámetros del sistema que se describen a continuación:

- *Frecuencia del sistema:* La frecuencia del sistema es la frecuencia de Clock a la cual correrá el procesador y el bus PLB, en el diseño se eligió una frecuencia de 75 Mhz. La elección de esta frecuencia determinará la velocidad de procesamiento total del sistema, así como también la velocidad de comunicación con los periféricos. Como desventaja, aumentar la frecuencia del sistema trae consigo un aumento en el consumo de potencia del sistema.
- *Memoria local:* La memoria local, es la memoria externa al procesador de acceso más rápido, y está compuesta por BlockRAMs, se asignaron 8KB a cada procesador como se muestra en la Figura 24. El procesador accede a ellas mediante el bus LMB. Se debe aclarar que el procesador MicroBlaze posee una arquitectura de memoria Harvard, por lo que accede a la memoria de instrucciones y de datos por separado (existen dos mapas de memoria distintos).

- *Tipo de bus de periféricos:* El bus del Sistema puede seleccionarse entre el bus PLB y el bus AXI. En este trabajo se optó por el bus PLB, ya que el bus AXI es el nuevo estándar, y no es compatible con muchas IP actualmente disponibles.

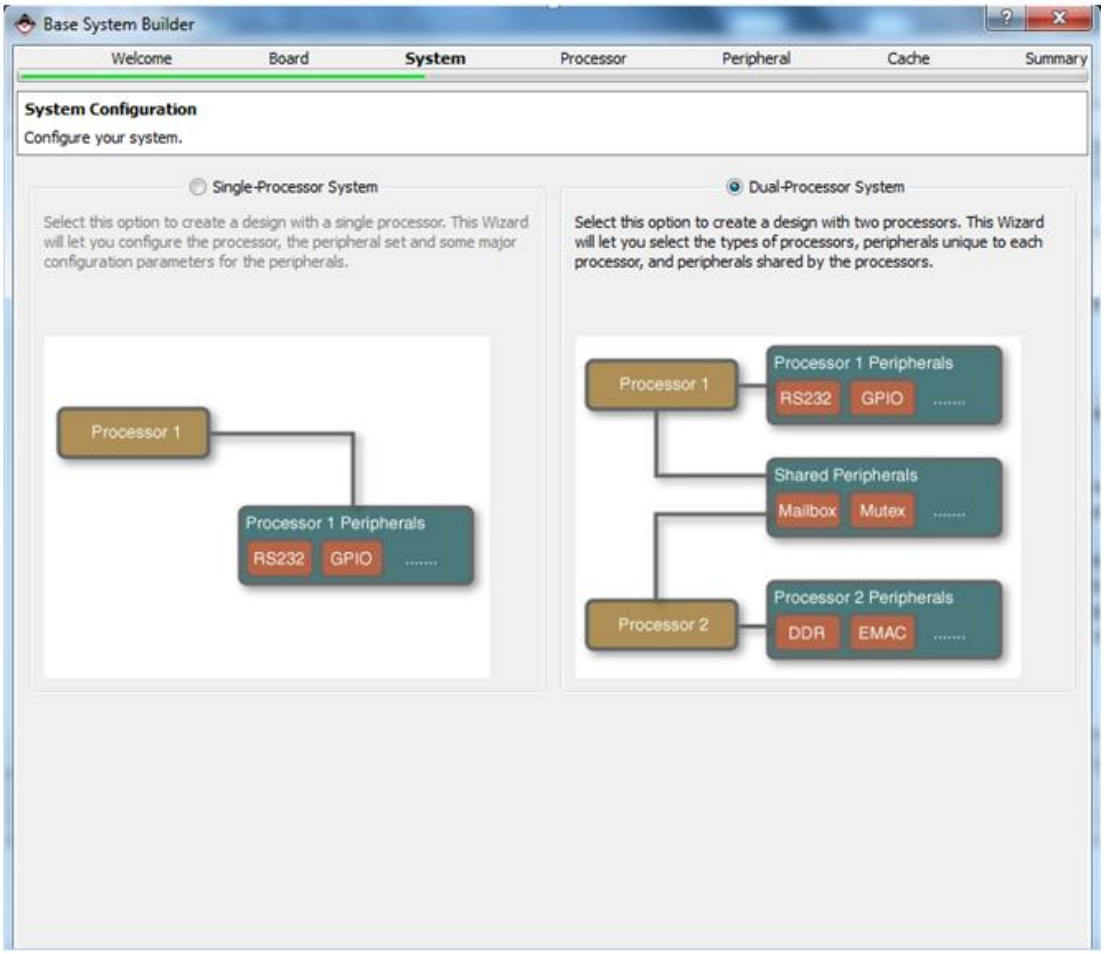

Figura 23. Selección de sistema de dos procesadores

Parámetros de implementación del procesador

Los siguientes parámetros fueron elegidos para la implementación de cada procesador MicroBlaze, tal como se muestra en la figura 25.

• Frecuencia de reloj de referencia 100MHz
• Frecuencia de Operación 75Mhz. Pipeline de 3 Etapas.
• 8K de memoria local de Instrucción o Datos.
• Depuración (Debugger) Habilitada.
• Una unidad de Barrel Shifter.
• Una Unidad Multiplicadora de Enteros de 32-bits.
• Una Unidad de División de Enteros.
• Ninguna unidad de Punto Flotante.
• Sin "Branch Prediction" (Predicción de Saltos).

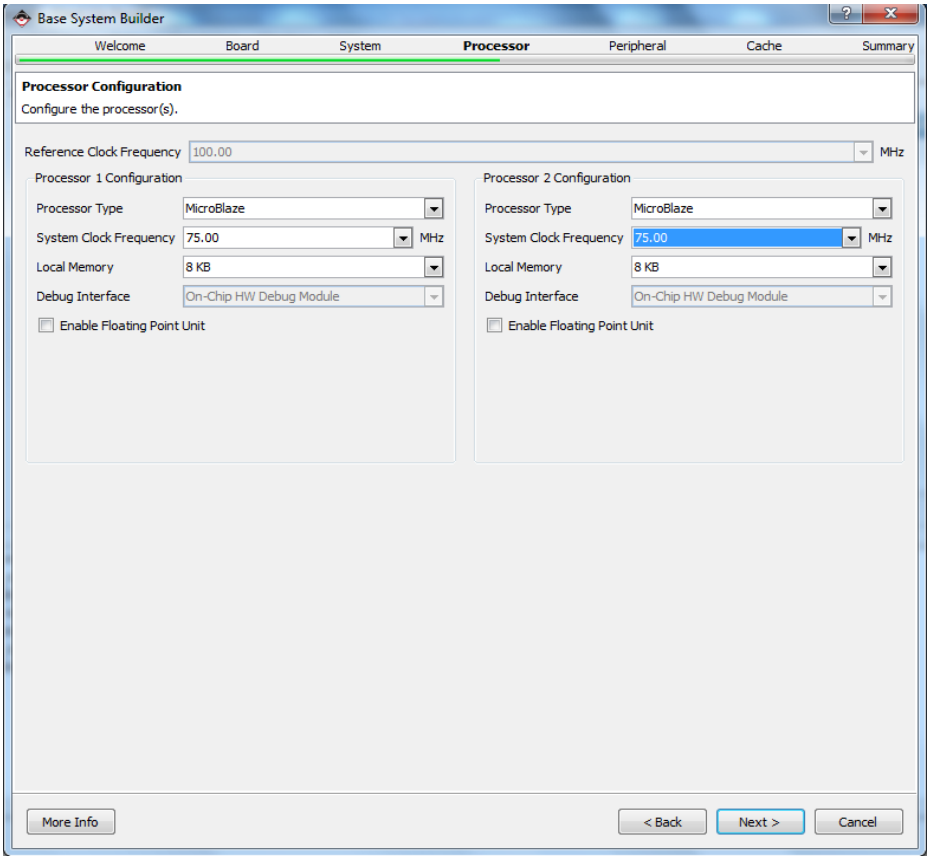

Figura 24. Parámetros del procesador

Ensamblado del sistema

La Figura 25 muestra los periféricos incorporados al sistema multiprocesador, entre ellos se debe destacar la incorporación de 128MB de memoria compartida DDR2-SDRAM, la interfaz UART para la recepción de datos desde el PC-Host, salidas digitales para medir tiempos de referencia, mailbox y mutex para la comunicación entre procesadores.

En la Figura 26 se muestra el ensamblado del sistema (System Assembly View) en XPS. En esta etapa se pueden agregar nuevos Ip Cores y sus interfaces con los procesadores. En esta vista se pueden observar todos los Cores correspondientes al Sistema (en la derecha) y sus buses de conexiones (en la izquierda).

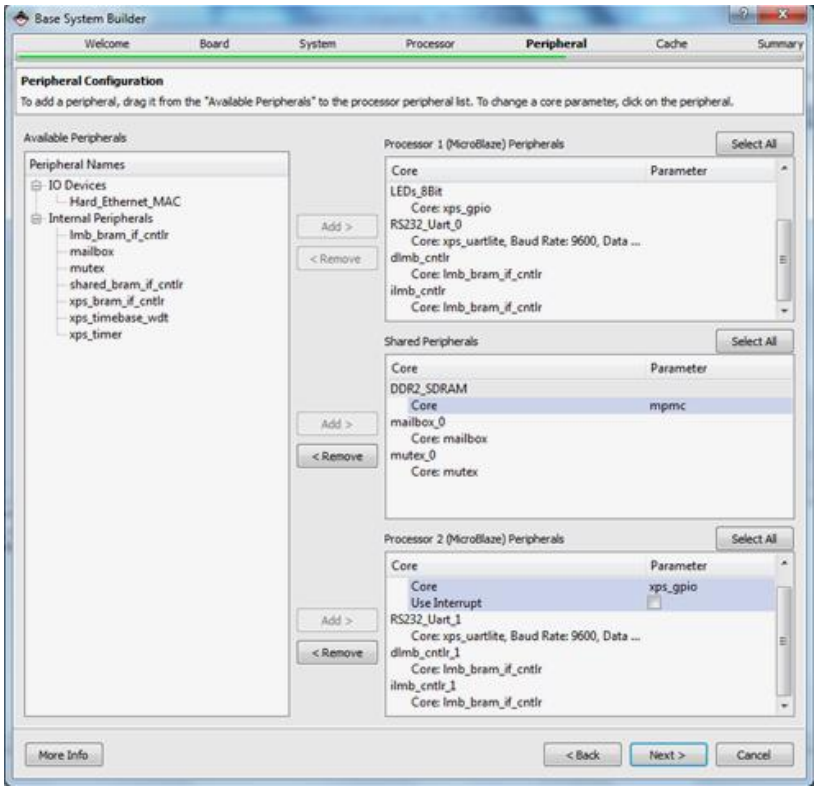

Figura 25. Selección de periféricos de cada Microblaze

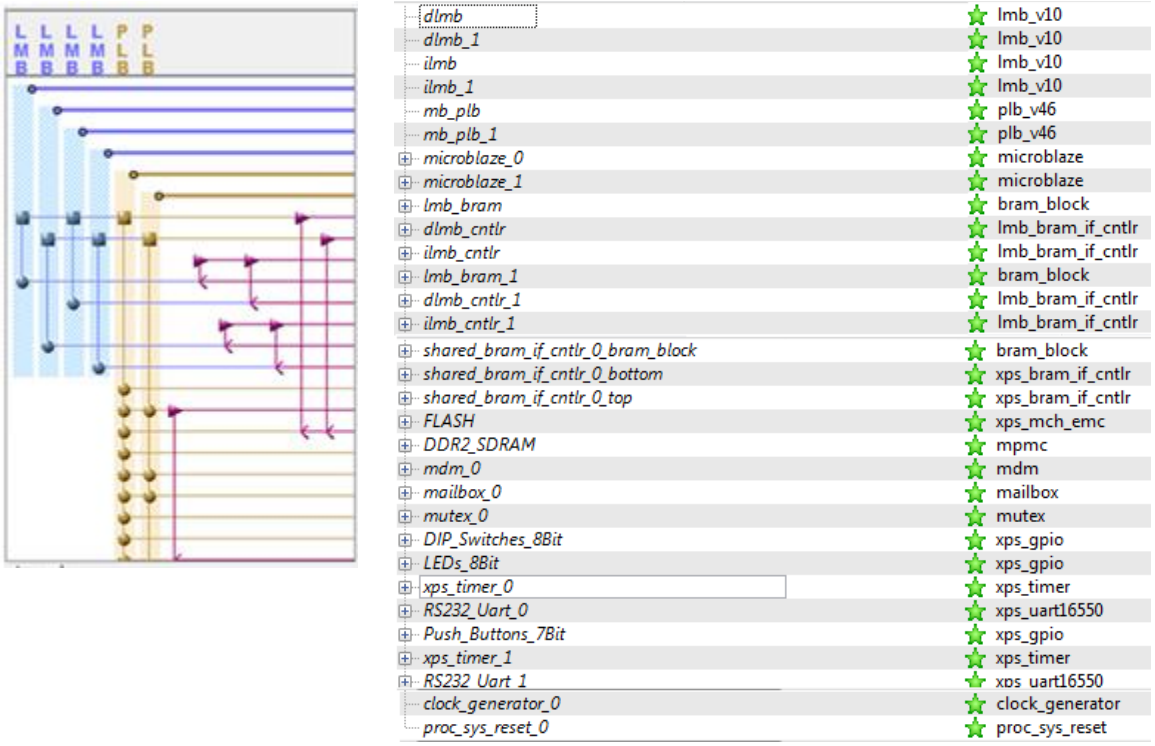

Figura 26. Ensamblado del sistema

Buses y periféricos

Como se puede ver en la Figura 26, existen 6 buses en total en el Sistema multiprocesador. El bus de periféricos principal, es del tipo PLB, y la instancia en el ensamblado del sistema se llama mb_plb para el MicroBlaze_0 y mb_plb1 para el MicroBlaze_1. Contiene dos maestros que son los procesadores, y varios periféricos. Las instancias de los periféricos son llamadas: "mdm_0" (MicroBlazeDebug Module), "xps_timer_0/1" (temporizadores), LEDs_8bits y "DIP_switches_8Bit" (Core de los Pulsadores), Push_Buttons_7Bits, etc.

Los buses de Memoria son dos, uno para instrucción y otro para datos, del tipo LMB. Los nombres de las instancias son "dlmb", para memoria de datos, e "ilmb" para memoria de instrucciones. Cada uno contiene asociado un controlador ("dlmb_cntlr" e "ilmb_cntlr"), y ambos acceden a una instancia de Block RAM ("lmb_bram").

Además, existen dos instancias más, las cuáles no tienen conexiones de bus. Una de ellas es "clock_generator_0", que se encarga de tomar el Clock de entrada a la FPGA (100Mhz) y generar los Clocks que necesitará el sistema mediante un DCM o PLL (en este caso solamente generará 75 Mhz para los procesadores y el bus PLB).

La otra instancia es "proc_sys_reset_0", que se encarga de sincronizar el reset proveniente del pulsador en la Placa, al Clock del Sistema, y consecuentemente propagar la señal de reset a todos los módulos.

El microblaze 0 tiene conectados los siguientes periféricos:

- leds 8bit
- dip_Switches
- uart 1 9600 baudios
- controlador de memoria BRAM

El microblaze 1 tiene conectados los siguientes periféricos:

- uart 2
- Pulsadores
- controlador de memoria BRAM

Además, se dispone de los siguientes periféricos compartidos:

- mailbox y mutex
- ddr2-SDRAM

En este diseño no se agregó cache a los procesadores para mantener la coherencia con la memoria compartida.

Mapa de direcciones

La Figura 27 muestra el mapa de direcciones final del microblaze 0.

microblaze_0's Address Map						
dlmb_cntlr	C_BASEADDR	0x00000000	0x0000FFFF	64K	SLMB	dlmb
ilmb_cntlr	C_BASEADDR	0x00000000	0x0000FFFF	64K	SLMB	ilmb
LEDs_8Bit	C_BASEADDR	0x81400000	0x8140FFFF	64K	SPLB	mb_plb
DIP_Switches_8Bit	C_BASEADDR	0x81420000	0x8142FFFF	64K	SPLB	mb_plb
FLASH	C_MEM0_BASE...	0x82000000	0x82FFFFFF	16M	SPLB	mb_plb
xps_timer_0	C_BASEADDR	0x83C00000	0x83C0FFFF	64K	SPLB	mb_plb
RS232_Uart_0	C_BASEADDR	0x83E00000	0x83E0FFFF	64K	SPLB	mb_plb
mdm_0	C_BASEADDR	0x84400000	0x8440FFFF	64K	SPLB	mb_plb
shared_bram_if_cntlr_0_top	C_BASEADDR	0x85F08000	0x85F0FFFF	32K	SPLB	mb_plb
DDR2_SDRAM	C_MPMC_BASE...	0x88000000	0x8FFFFFFF	128M	SPLB0:SPLB1	mb_plb:mb_plb_1
mutex_0	C_SPLB0_BASE...	0xC2400000	0xC240FFFF	64K	SPLB0	mb_plb
mailbox_0	C_SPLB0_BASE...	0xCDE00000	0xCDE0FFFF	64K	SPLB0	mb_plb

Figura 27. Direcciones del Sistema Embebido

La Figura 28 muestra el mapa de direcciones final del microblaze 1.

microblaze_1's Address Map						
dlmb_cntlr_1	C_BASEADDR	0x00000000	0x0000FFFF	64K	SLMB	dlmb_1
ilmb_cntlr_1	C_BASEADDR	0x00000000	0x0000FFFF	64K	SLMB	ilmb_1
Push_Buttons_7Bit	C_BASEADDR	0x81400000	0x8140FFFF	64K	SPLB	mb_plb_1
xps_timer_1	C_BASEADDR	0x83C00000	0x83C0FFFF	64K	SPLB	mb_plb_1
RS232_Uart_1	C_BASEADDR	0x83E00000	0x83E0FFFF	64K	SPLB	mb_plb_1
shared_bram_if_cntlr_0_bot...	C_BASEADDR	0x85F08000	0x85F0FFFF	32K	SPLB	mb_plb_1
DDR2_SDRAM	C_MPMC_BASE...	0x88000000	0x8FFFFFFF	128M	SPLB0:SPLB1	mb_plb:mb_plb_1
mutex_0	C_SPLB1_BASE...	0xC2400000	0xC240FFFF	64K	SPLB1	mb_plb_1
mailbox_0	C_SPLB1_BASE...	0xCDE00000	0xCDE0FFFF	64K	SPLB1	mb_plb_1

Figura 28. Direcciones del Sistema Embebido

Conexiones externas

En el diagrama de la Figura 29 se muestra el esquema de periféricos externos al sistema, que permiten la comunicación con los módulos exteriores.

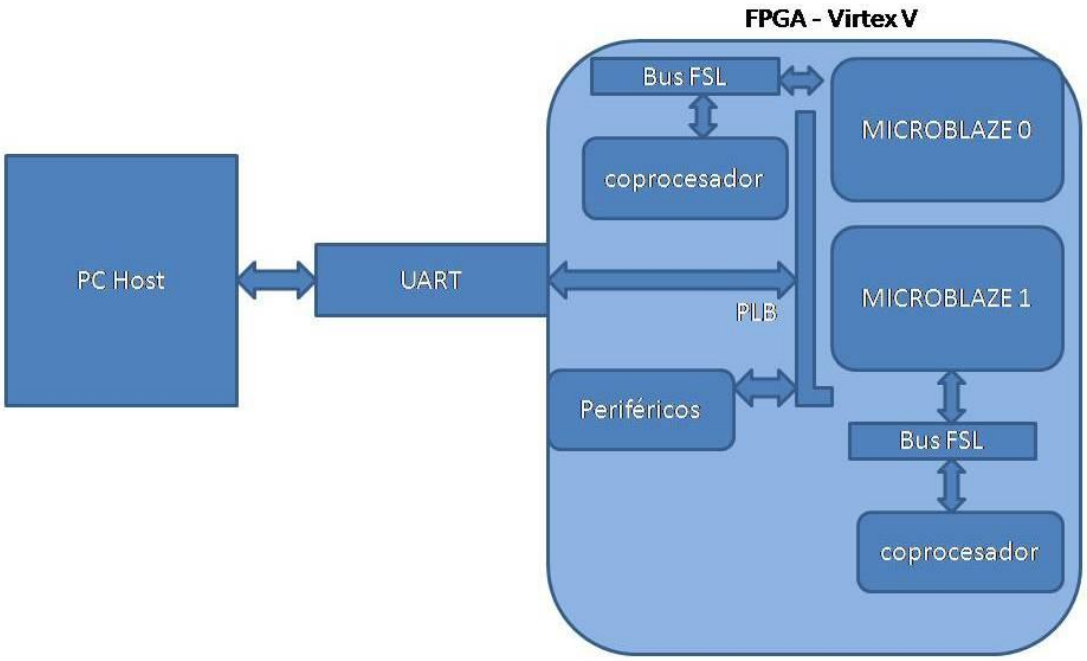

Figura 29. Conexiones del sistema completo

Utilización de recursos de la FPGA

Aunque la FPGA tiene una amplia variedad de recursos y algunos de estos fueron incorporados solo para testeo, se considera necesario hacer un análisis de los recursos utilizados para el sistema multiprocesador, que se muestran en la tabla 1. Las slices son un buen indicador de la utilización de la FPGA, ya que sin slices no se pueden agregar nuevas funcionalidades al diseño (aunque se disponga de otros recursos). El indicador de slices utilizados diferencia entre las ocupadas parcialmente y totalmente, esto significa que se podrían seguir utilizando LUTs o registros en las slices parcialmente ocupadas.

Tabla 1. Utilización de slices

Utilización lógica de Slice	Usadas	Disponibles	Porcentaje ocupado
Número de Registros Slice	10.202	28.800	35%
Número de LUTs Slice	9.798	28.800	34% (32% para lógica)
Número usado como memoria	427	7.680	5%
Número de Slice ocupados	5.057	7.200	70%
Número de BlockRAM/FIFO	57	60	95%
Memoria total usada (KB)	2.052	2.160	95%
Número de DSP48Es	10	48	20%

También es importante tener en cuenta los bloques E/S o IOB, donde se ocupan 189 de 480 (un 39%) para el sistema completo. Se debe tener en cuenta que en el sistema no se intenta optimizar la ocupación del espacio, debido a que la mayoría de los periféricos conectados a los procesadores son utilizados para testeo, pero se tiene en cuenta el espacio utilizado por los procesadores y los coprocesadores para determinar la eficiencia en el espacio consumido por el sistema. Además, se debe aclarar que los tamaños de memoria han sido sobredimensionados, esto quiere decir que el procesamiento de las imágenes se puede hacer con mucho menos espacio de memoria, posibilitando así la implementación de mayor cantidad de procesadores microblaze con los mismos recursos.

Para diferenciar los recursos indispensables para el sistema, en la Tabla 2 se muestras los recursos utilizados por cada componente del sistema. Aquí interesan los componentes utilizados por los procesadores, en donde se utiliza un 33% de los flip-flops y un 45% de las LUTs aproximadamente. Además, como se mencionó anteriormente el uso de la memoria DDR2 y BRAM se puede reducir considerablemente para la aplicación implementada.

Tabla 2. Resumen de la Síntesis XPS

Reporte	Flip-Flops usados	LUTs usados	BRAMS usados
Sistema	12588	10766	49
MicroBlaze 1	2264	2281	
MicroBlaze 0	2264	2281	
UART 1	387	395	
UART 0	387	395	
MUTEX 0	145	90	
MAILBOX0	350	317	
Bram compartida	232	264	8
Timer 0 y Timer 1	358 c/u	291 c/u	
flash	477	345	
ddr2 sdram ()	4441	2500	9

Lmb bram 0			16
Lmb bram 1			16
plb	162	501	
plb_1	156	409	

4.3.2.2. Software del procesador

El Software del procesador fue escrito en código C, en el entorno de Xilinx SDK [19]. El programa involucra la integración de todos los módulos del Sistema Embebido, así como la funcionalidad de los algoritmos de procesamiento de imágenes.

Inicialización

Al comenzar la ejecución, el procesador realiza varias rutinas de inicialización.

- Inicializar los Cores de Entrada/Salida: El Core GPIO es el encargado de controlar los pulsadores que se encuentran en la placa. La rutina de inicialización habilita el Core, y lo configura como entrada para leer el estado de los pulsadores. Además, se inicializan las salidas conectadas a los leds que también servirán de referencia para medir el tiempo de ejecución de cada algoritmo.
- Inicializar los módulos UART: Se deben inicializar las interfaces UART para la comunicación con el PC-Host
- Inicializar los controladores de mailbox y memoria compartida.

Una vez finalizadas las rutinas todos los componentes del sistema embebido quedan configurados para ser usados.

Bucle principal

El procesador MB0recibirá los datos de la imagen por puerto serie desde el PC-Host para almacenarlos en memoria compartida. A partir de aquí se realizarán todos los pasos de procesamiento y almacenamiento de resultados.

Las distintas etapas del programa se describen a continuación:

- Almacenamiento en memoria compartida: Esta etapa es solo para la recepción de los píxeles desde la PC y el almacenamiento en memoria compartida, por lo que no será tenida en cuenta para la evaluación de desempeño del sistema. Previo a la ejecución del algoritmo de procesamiento, se inicializa una salida en alto para la medición del tiempo de ejecución mediante un osciloscopio. Los detalles del código para la recepción, almacenamiento, procesamiento y transmisión de los datos relativos al MB0, se muestran en el Apéndice C sección 1.

- Bucle de ejecución del algoritmo de procesamiento de imágenes: Esta etapa fue implementada de tres formas diferentes, por un solo procesador, por dos procesadores con y sin coprocesador. Para la ejecución con dos procesadores, el bucle contempla el sincronismo mediante pasaje de mensajes con el mailbox para indicar al MB1 cuando

comenzar la lectura de memoria y el procesamiento de las ventanas de píxeles (Los detalles de código correspondientes al MB1 se muestran en el Apéndice C sección 2). Para el acceso a memoria compartida no es necesario usar un mutex debido a que cada procesador accede a diferentes partes de la memoria por diferentes puertos, posibilitando el acceso simultáneo. En cada iteración del bucle se lee una nueva fila de píxeles, (utilizada con las dos filas previamente leídas para formar las ventanas de 3x3), que permite obtener como resultado del procesamiento una fila de salida, la cual es almacenada en otro sector de la memoria compartida para evitar la necesidad de sincronismo durante el almacenamiento.

- Devolución de la imagen procesada: Al finalizar el bucle, el procesador MB1 debe informar al MB0 la finalización del procesamiento para que este ponga a cero lógico el pin de salida utilizado para temporización y realice la transmisión hacia el PC-Host.

- Pulsadores: Los pulsadores solo serán utilizados para coordinar el inicio de procesamiento con el PC-Host.

Comunicación entre Procesadores

Para la comunicación entre procesadores se utiliza el mailbox, que permite enviar mensajes de un procesador a otro. Este método es muy simple y consiste en la utilización de un conjunto de funciones para la lectura y escritura de mensajes.

Las funciones más importantes de mailbox son:

- XMbox_LookupConfig(MboxDeviceId)

Buscar los datos de configuración en la tabla de configuración del dispositivo. Esta información de configuración se utiliza a continuación para inicializar este componente.

- XMbox_CfgInitialize(&Mbox, ConfigPtr, ConfigPtr->BaseAddress)

Aquí se realiza el resto de la inicialización

- MailboxExample_Send(&Mbox, CPU_ID)

Envía un mensaje que tiene como destinatario al procesador indicado en CPU_ID

- MailboxExample_Receive(&Mbox, CPU_ID)

Recibe un mensaje que tiene como remitente al procesador indicado en CPU_ID

Comunicación con el Co-procesador

Desde el punto de vista del procesador es muy simple acceder al co-procesador (Bloque de ordenamiento/convolución/promediado en VHDL). El procesador tiene instrucciones dedicadas para enviar y recibir palabras del bus FSL.

Las más importantes de estas instrucciones son:

- getfslx(val,id,flags)

Realiza la tarea de obtener una palabra del bus FSL. El argumento "val" es una variable del programa que es el destino de la palabra obtenida del bus FSL. "id" es el identificador del bus FSL. "flags" son banderas que indican la forma en la que se ejecuta la instrucción.

- putfslx(val ,id , flags)

Realiza la tarea de enviar una palabra hacia el bus FSL de salida. El argumento "val" es una variable del programa que es la fuente de la palabra a enviar al bus FSL. "id" es el identificador del bus FSL. "flags" son banderas que indican la forma en la que se ejecuta la instrucción.

Los "flags" mencionados afectan la ejecución de estas instrucciones, entre los "flags" más importantes se destacan:

- FSL_DEFAULT: Hace que la función de FSL sea bloqueante, es decir, que la instrucción no termine hasta que la operación deseada haya terminado.
- FSL_NONBLOCKING: Hace que la función de FSL sea no-bloqueante, es decir, que la instrucción terminará independientemente si la operación se llevó a cabo exitosamente o no.
- FSL_CONTROL: Hace que la función de FSL sea de Control, es decir envía o recibe datos de control del bus.
- FSL_NONBLOCKING_CONTROL: Hace que la función de FSL sea de Control y no-bloqueante, es decir envía o recibe datos del bus que tengan el bit de control en "1". Al ser no-bloqueante, la instrucción terminará independientemente si la operación se llevó a cabo exitosamente o no.

Interfaz FSL con el bloque de ordenamiento/convolución/promediado de ventanas:

Para obtener un píxel de salida, el procesador envía un paquete de datos de 32-bits mediante el bus FSL correspondiente al bloque de operadores de ventana. Esto lo hace mediante la instrucción putfslx(val,id,flags).

El procesador recibirá el resultado del procesamiento de píxeles, mediante la instrucción getfslx(val,id,flags), para el almacenamiento en memoria compartida.

El bloque coprocesador recibirá las palabras a procesar de 32-bits al siguiente ciclo, cuando la FIFO que administra los datos del bus le indique que recibió una palabra (mediante una bandera de "empty"[1]). El envío del dato procesado al procesador se realiza en un solo ciclo.

[1]La bandera de "empty" ("vacío") es un bit que provee una FIFO, para indicar que está vacía cuando está activa (o lo que es equivalente que –no- está vacía cuando está desactivada).

5. Cálculo de rendimiento y resultados

El diseño de hardware para los algoritmos de procesamiento de imágenes basados en operadores de ventana de 3x3 se sintetizó para las arquitecturas de la FPGA Virtex V de Xilinx [24]. La FPGA de Xilinx proporcionó una implementación rápida y eficiente, tal como se esperaba. La tabla 3 muestra los resultados para la síntesis del diseño ordenador.vhd y convolucion.vhd. Se debe aclarar que la síntesis del filtro de media proporcionó una utilización de recursos despreciable respecto a los recursos utilizados por el sistema, es por eso que no se tuvo en cuenta en el análisis de recursos utilizados.

Tabla 3. Recursos utilizados para la síntesis de ordenamiento y convolución en VHDL

FPGA VIRTEX de XILINX	% DE MEMORIA	% DE LÓGICA	total
Ordenamiento	670 (2,32%)	704 (2,45 %)	28800
Convolución	474 (1,65%)	323 (1,13%)	28800

5.1. Procesos VHDL en testbench

Con el fin de examinar el código VHDL para la correcta funcionalidad, las herramientas de xilinx ofrecen una función denominada simulación. La Simulación toma el código VHDL y simula cómo funcionaría en hardware. Para esto, el diseñador debe proporcionar al simulador entradas válidas para producir resultados esperados. Un método eficiente y común de simular código VHDL es mediante el uso de un tipo especial de código VHDL llamado banco de pruebas (testbench). Los bancos de prueba cubren eficientemente el código VHDL que el diseñador desea simular y también proveen estímulos para la entidad testeada.

Un banco de pruebas de un algoritmo es responsable de estimular las señales de entrada esenciales para aquel algoritmo. Ya que los diseños utilizados en este enfoque son todos sincrónicos, se debe proveer una señal de reloj. Además, todos los algoritmos diseñados proporcionan una funcionalidad de reset que permite reiniciar los algoritmos en cualquier punto. En la Figura 30 se muestra el testbench del algoritmo de ordenamiento. El testbench de la convolución se puede observar en la Figura 31 ([25] y [26]).

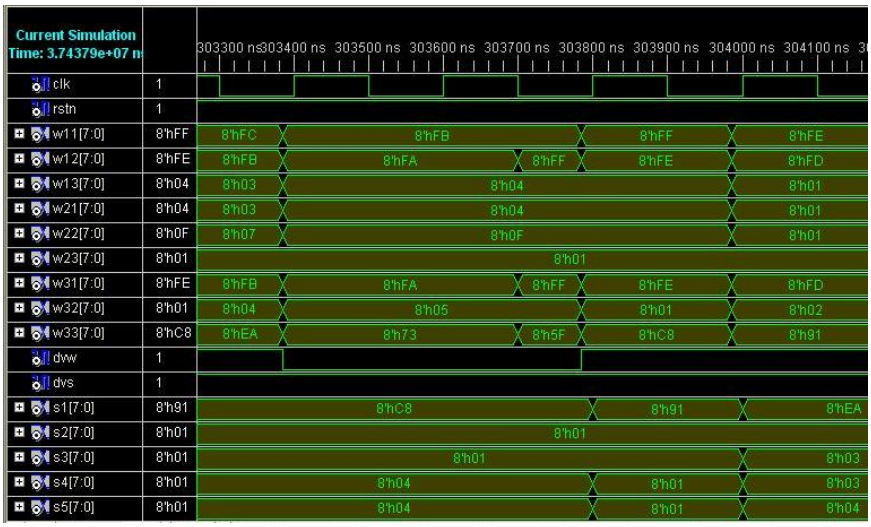

Figura 30. Testbench del algoritmo de ordenamiento

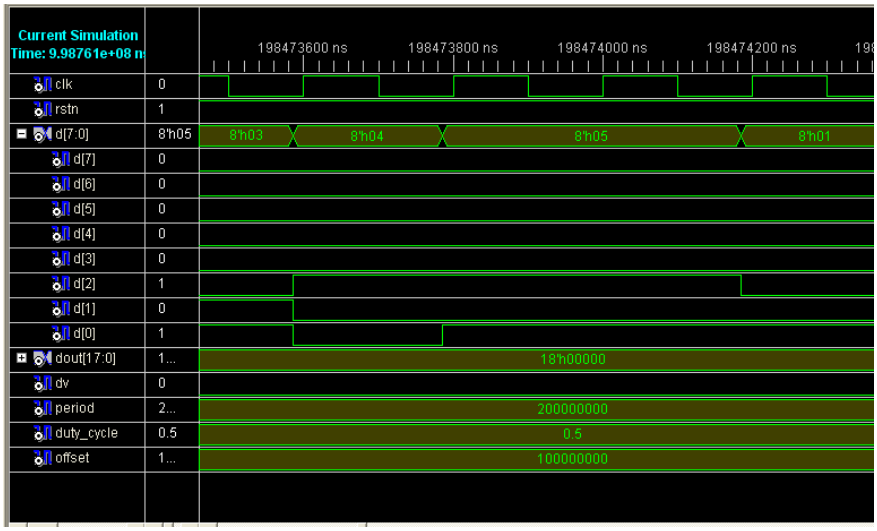

Figura 31. Testbench de la convolución

5.2. Tiempos de Procesamiento

El programa que se ejecuta en el primer procesador es ligeramente diferente al que se ejecuta en el segundo, principalmente contiene las líneas de código que se utilizan para sincronizar el inicio del procesamiento y el envío de los datos procesados a la PC-Host mediante el mailbox. El resto de las líneas de código corresponden a la ejecución del algoritmo.

Aunque el objetivo principal de la tesis consiste en la optimización del desempeño en la ejecución de algoritmos de procesamiento de imágenes mediante un sistema multiprocesador, es importante plantear la forma más eficiente de transmisión y procesamiento de datos que deberá ser tenida en cuenta durante la implementación de las tareas a futuro. Entonces, la mejor forma de aprovechar más eficientemente los recursos de los procesadores consiste en implementar un esquema de pipeline, iniciando la ejecución del algoritmo en el procesador MB1 mientras el procesador MB0 termina de almacenar los datos recibidos en memoria compartida. Cuando el Procesador MB0 termina de recibir los datos puede comenzar con el

procesamiento de los datos correspondientes, aunque dependiendo de la velocidad de transferencia utilizada para la recepción de datos, se podría configurar el MB0 para que se ejecute el algoritmo a medida que se van recibiendo los datos (para esto debería modificarse el esquema de procesamiento entre MB0 y MB1). Para obtener mayor eficiencia en la implementación del pipeline de transferencia y procesamiento, es conveniente que los tiempos de cada tarea sean similares, para lograr esto se debería utilizar un protocolo de comunicaciones que permita una tasa de transferencia mucho mayor.

En la figura 32, se muestra el esquema con las respectivas tareas, en donde R representa recepción desde la PC-HOST, A el almacenamiento en memoria compartida, L lectura de memoria compartida, P procesamiento, E escritura de datos procesados y T transmisión al PC-Host. En esta plataforma de procesamiento de imágenes la lectura de datos de memoria compartida consume 83 useg por fila; el procesamiento de datos para el filtro de media es de 230 useg por fila, para el de mediana/erosión/dilatación 3740 useg y para el de sobel 5080 useg; por su parte la escritura de una fila de datos consume 70 useg. Se debe tener en cuenta que la temporización fue realizada para imágenes de 128x128 píxeles, lo que quiere decir es que cada fila es de 128 píxeles de longitud.

Figura 32. Pipeline de recepción, procesamiento y transferencia

5.2.1. Estimación de tiempos usando Matlab

Para la transmisión de los píxeles y la medición del tiempo de procesamiento, inicialmente se utilizó matlab (ver códigos de transmisión, recepción y temporización en el apéndice c). Desde matlab se envían los datos mediante el puerto serie de la PC a 9600 baudios y se comienza con la cuenta de tiempo mediante los comandos TIC y TOC. En la Tabla 4 se muestran los tiempos de transmisión, almacenamiento, procesamiento y recepción de una imagen de 128x128 desde matlab [27]. Teniendo en cuenta que los tiempos de transmisión/recepción son mucho mayores a los de procesamiento y que la medición de tiempos desde el PC-host no es precisa debido a que influyen los tiempos propios del sistema operativo y de matlab, estos valores solo se tuvieron en cuenta como referencia del tiempo total de la ejecución, pero no para medir la eficiencia del sistema multiprocesador respecto a uno simple procesador.

Tabla 4. Tiempos totales de ejecución medidos desde Matlab

Algoritmo	Tiempos totales de transmisión, procesamiento y recepción (seg)
Filtro de mediana	102,563
Filtro de media	101,59
Filtro de Sobel	104,31
Erosión/dilatación	102,563

5.2.2. Medición de tiempos de procesamiento con uno y dos procesadores

Para medir solo los tiempos de lectura/escritura de la memoria compartida y procesamiento del algoritmo, se utilizó un método que requiere cambiar el estado de un pin, (configurado como salida), al comenzar el procesamiento y devolverlo al estado original al finalizar. Colocando la punta del osciloscopio sobre dicho pin y configurando el trigger se pueden medir los tiempos de forma correcta tal como se muestra a continuación.

Los tiempos de lectura, escritura y procesamiento se midieron para la mínima cantidad de filas, lo que permitió sacar conclusiones respecto a los tiempos individuales por fila y de la imagen completa. En la Figura 33 se obtuvo el tiempo de lectura de tres filas, que es la cantidad mínima necesaria para obtener las primeras ventanas válidas, y de escritura de una fila, que fue de 320 useg.

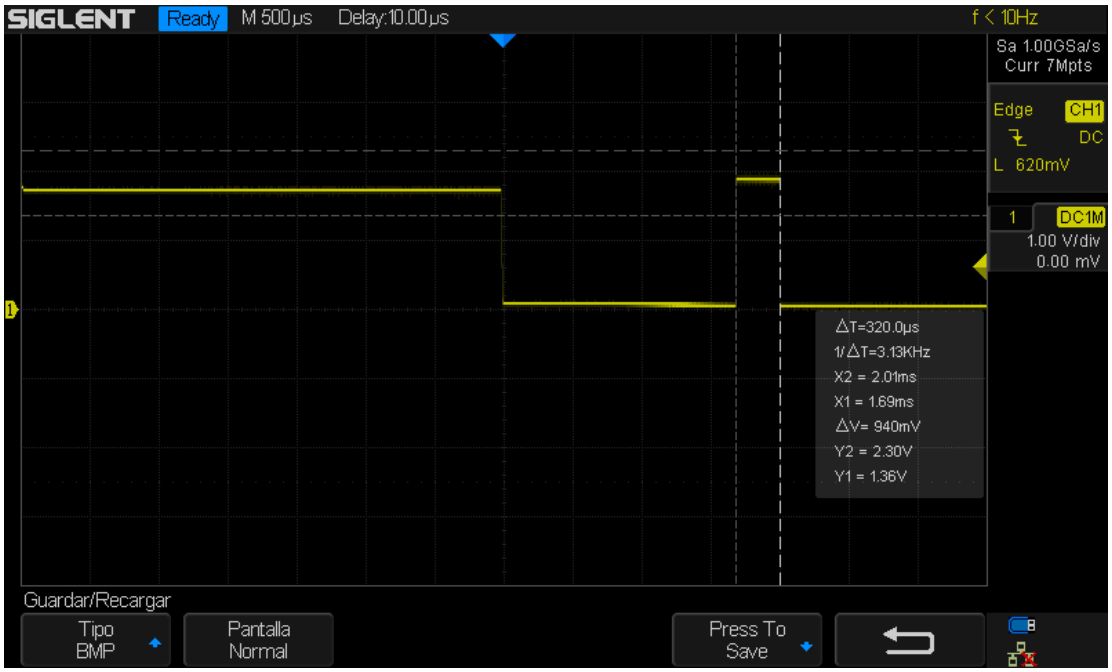

Figura 33. Tiempo de lectura de 3 filas y escritura de una fila sin procesamiento

En la Figura 34 se obtuvo el tiempo de lectura de las cuatro primeras filas, lo que permite determinar los tiempos de escritura y lectura necesarios para obtener las dos primeras filas procesadas, que fue de 470 useg. Hasta aquí, haciendo la diferencia entre los tiempos medidos se puede determinar el tiempo de lectura y escritura de una fila que es de 150 useg.

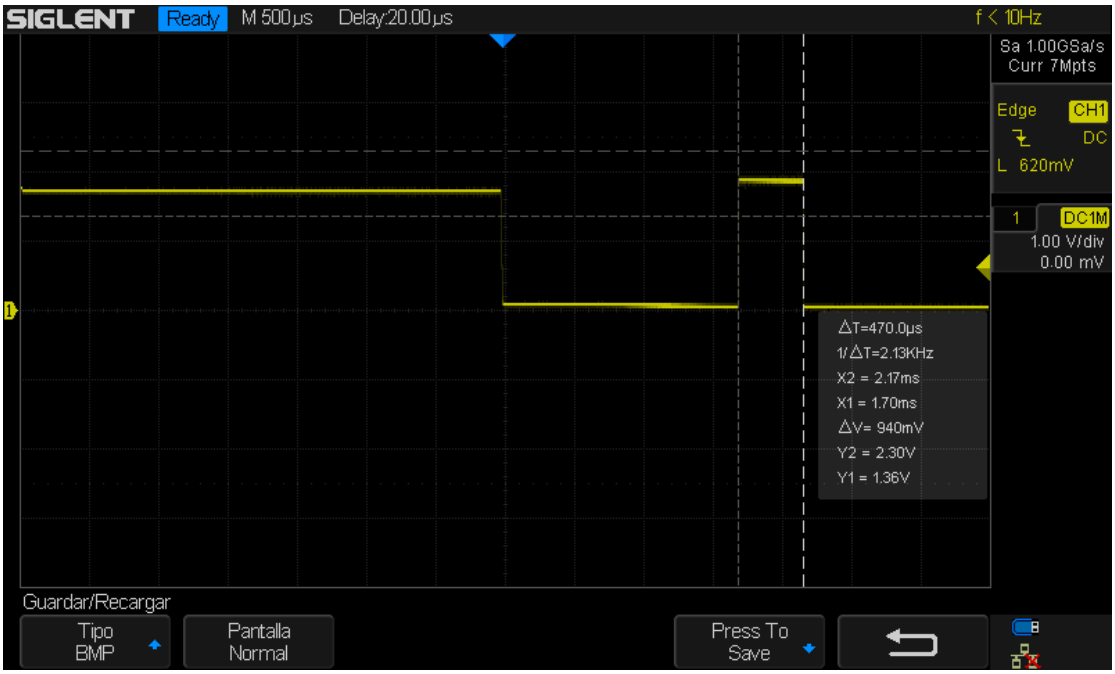

Figura 34. Tiempo de lectura de cuatro filas y escritura de dos filas sin procesamiento

De la Figura 35 se desprende que el tiempo de lectura de tres filas es de 250 us, haciendo la diferencia con el tiempo de lectura de tres filas y escritura de una se obtiene que la escritura de una fila desde memoria compartida le llevará 70 useg aproximadamente.

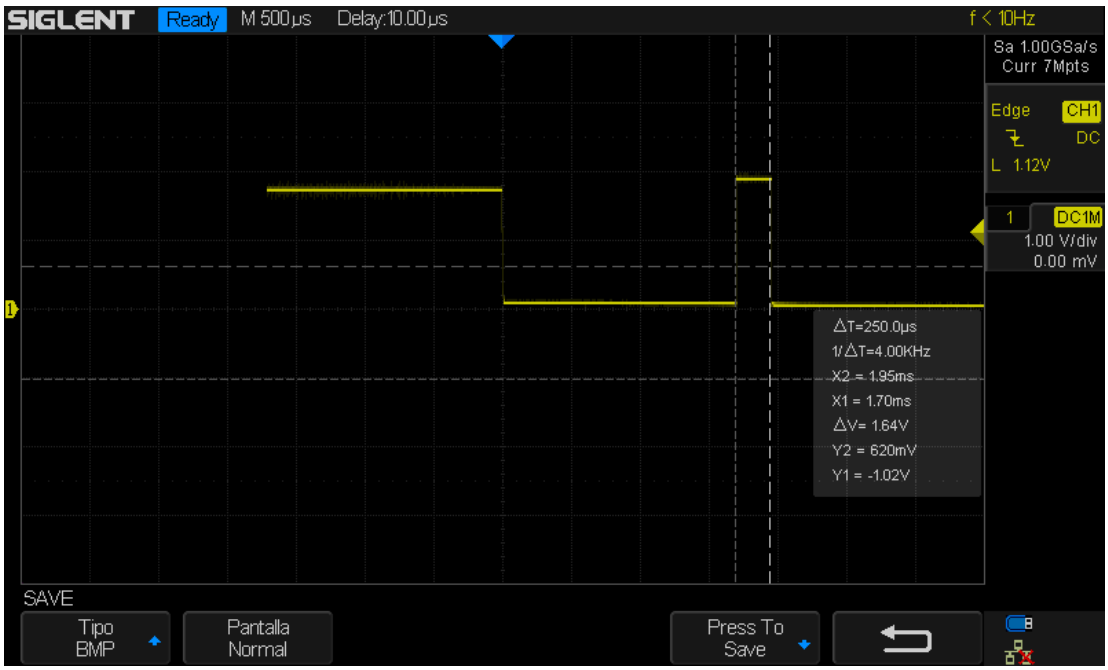

Figura 35.Tiempo de lectura de tres filas desde memoria compartida

En la Figura 36 se muestra el tiempo de envío y recepción de un mensaje de sincronismo mediante mailbox, que consume 12 useg. Este tiempo se puede considerar despreciable respecto al tiempo de lectura/escritura de una imagen completa, teniendo en cuenta que se

utiliza para indicar al MB1 que puede comenzar con el procesamiento y la respuesta es para indicar al MB0 que ya se terminó con el procesamiento.

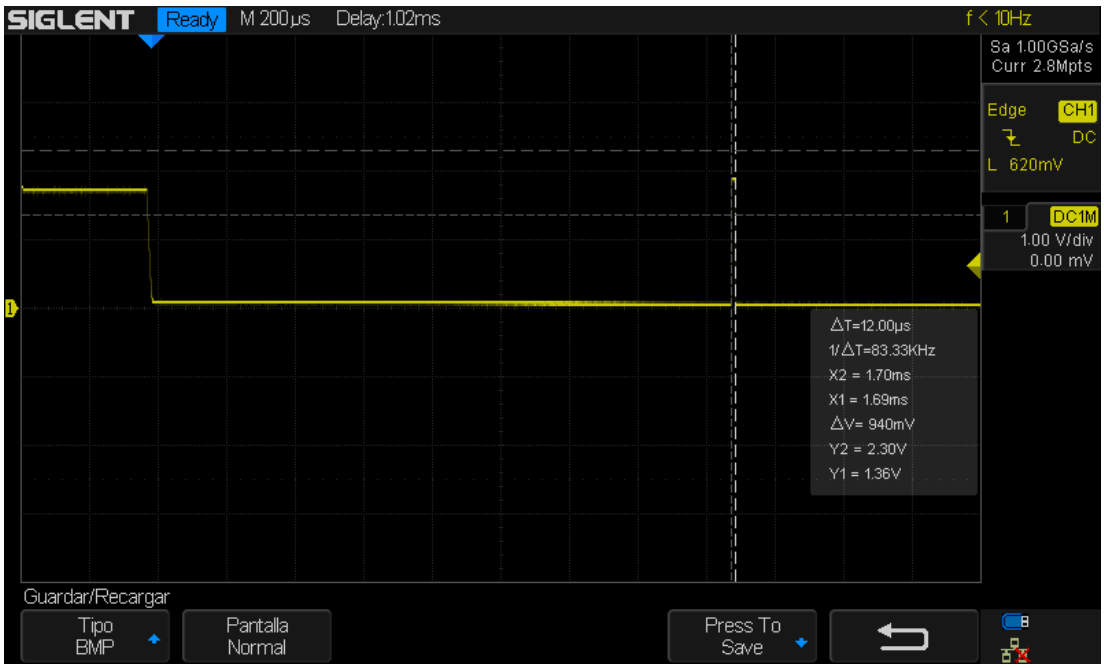

Figura 36.Tiempo de comunicación mediante mailbox entre MB0 y respuesta de MB1

En la Figura 37 se obtuvo el tiempo de lectura de tres filas, procesamiento del algoritmo de media y escritura de la fila resultado. Haciendo la diferencia con el tiempo que lleva solo la lectura y escritura, se obtiene que el tiempo de procesamiento necesario para la obtención de una fila es de 240 useg.

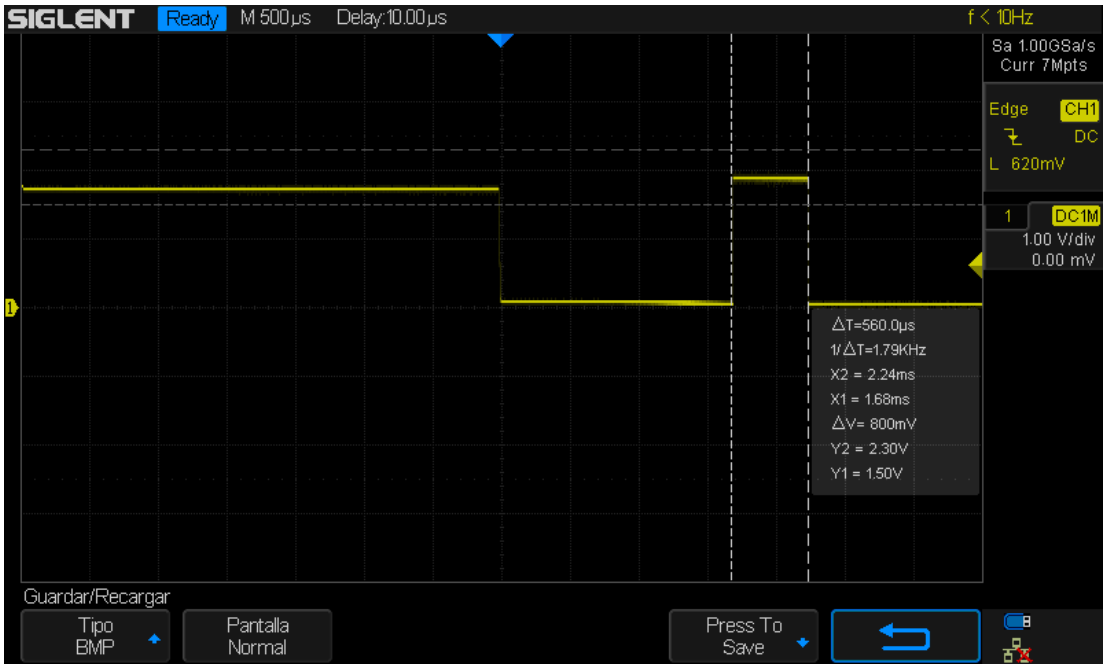

Figura 37. Tiempo de lectura de 3 filas, ejecución de algoritmo de media y obtención de una fila resultado

Para el caso de la lectura de cuatro filas para la obtención de dos filas de procesamiento, el tiempo total incluido el procesamiento es de 910 useg, menos los 470 useg del tiempo de lectura/escritura da 440 useg.

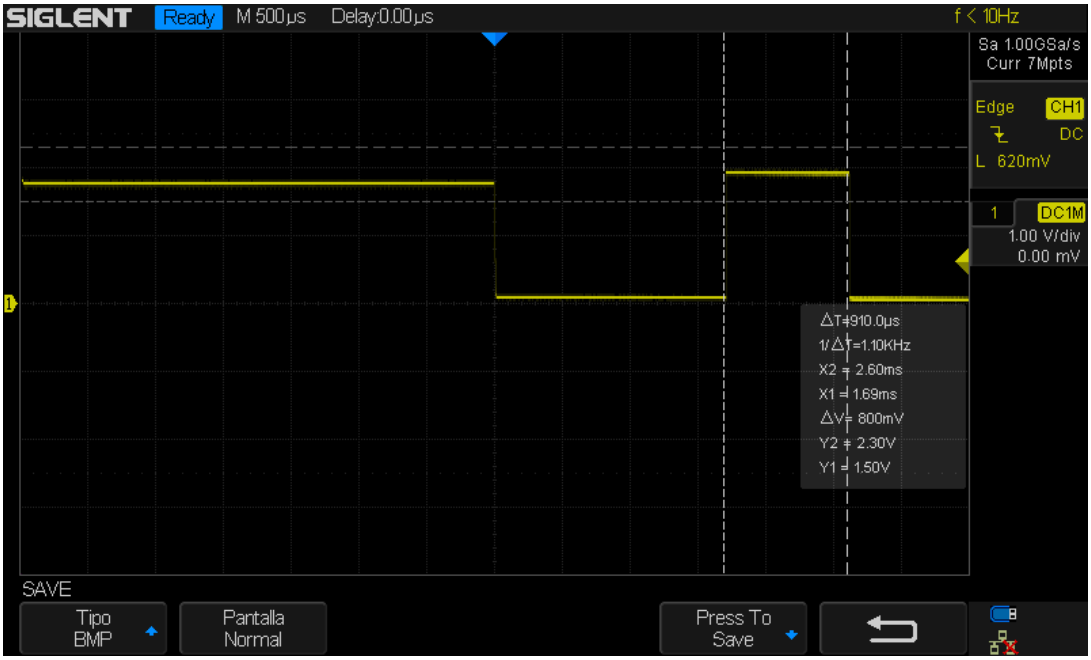

Figura 38. Tiempo de lectura de cuatro filas, ejecución de algoritmo de media y obtención de dos filas de salida

En la Figura 39 se obtuvo el tiempo de lectura de tres filas, procesamiento del algoritmo de mediana y escritura de la fila resultado. Haciendo la diferencia con el tiempo que consume realizar la lectura y escritura, se obtiene que el tiempo de procesamiento necesario para la obtención de una fila, al ejecutar el algoritmo de mediana, es de 3740 useg.

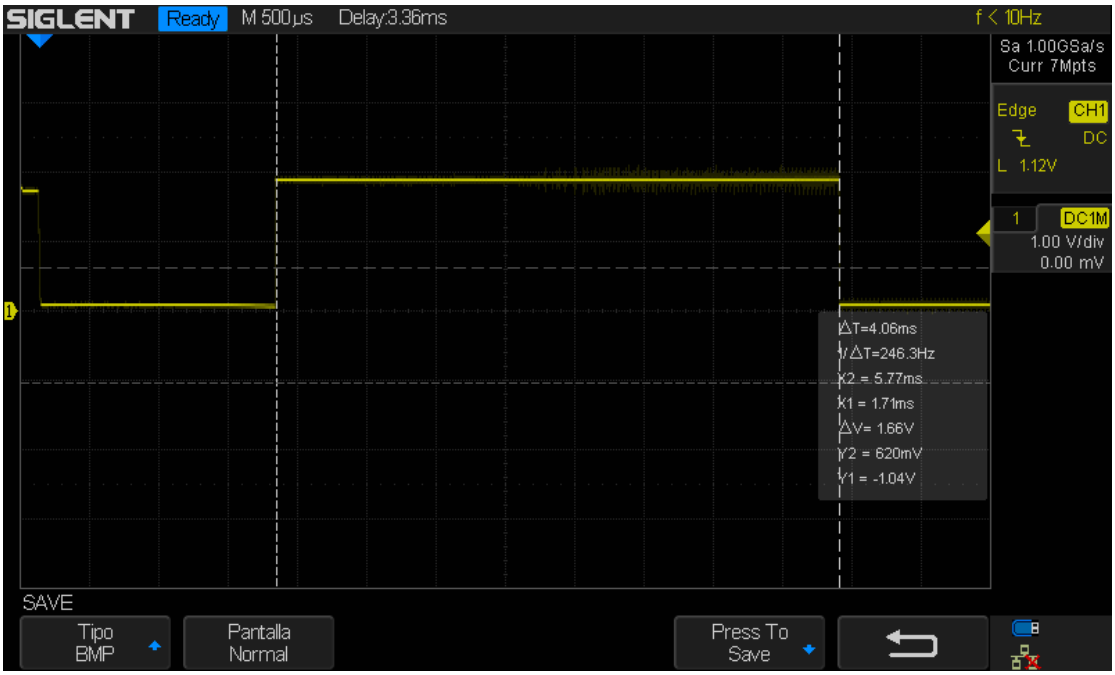

Figura 39.Tiempo de ejecución del filtro de mediana con tres lecturas, dos procesamientos y dos filas de salida

Para el caso de la lectura de cuatro filas para la obtención de dos filas de procesamiento, el tiempo total incluido el procesamiento es de 7960 useg, menos los 470 useg del tiempo de lectura/escritura da 7490 useg.

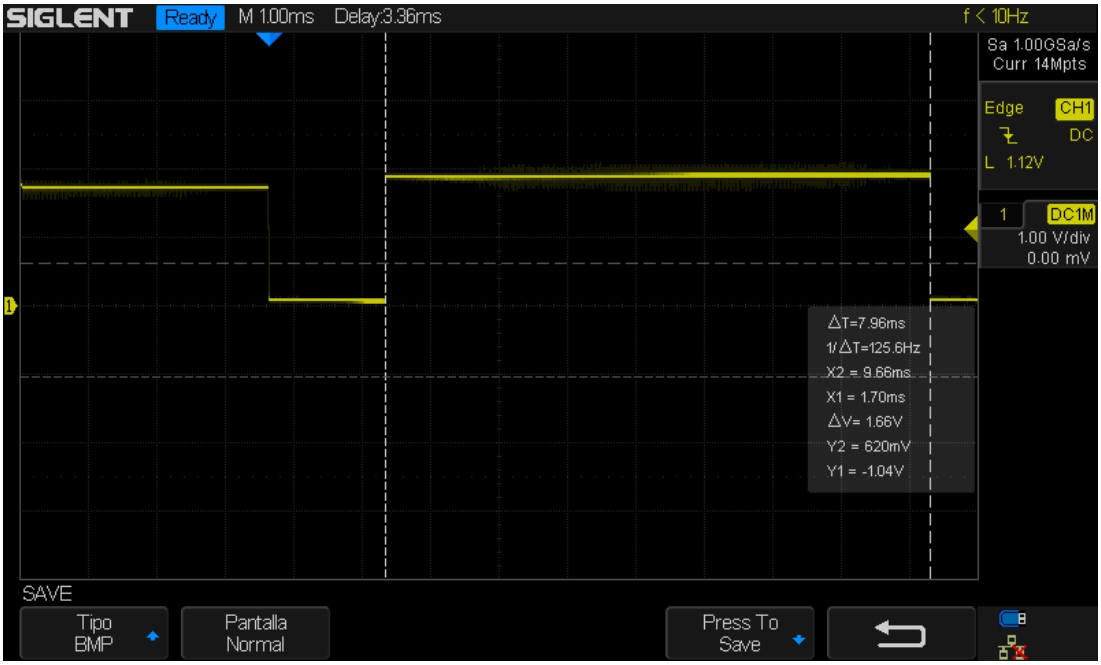

Figura 40.Tiempo de ejecución del filtro de mediana con cuatro lecturas, dos escrituras y dos procesamientos

En la Figura 41 se obtuvo el tiempo de lectura de tres filas, procesamiento del algoritmo de sobel y escritura de la fila resultado. Haciendo la diferencia con el tiempo que consume la lectura y escritura, se obtiene que el tiempo de procesamiento necesario para la obtención de una fila será de 6080 useg.

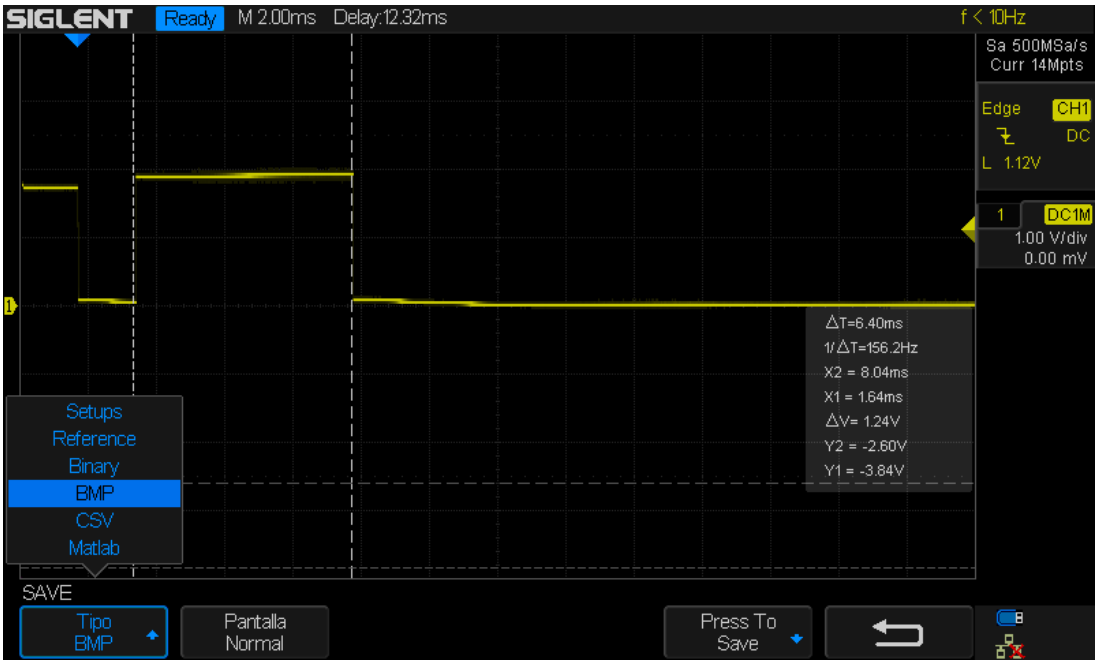

Figura 41. Tiempo de ejecución del filtro de sobel con tres lecturas, un procesamiento y una fila de salida

Para el caso de la lectura de cuatro filas para la obtención de dos filas de procesamiento del algoritmo de sobel, el tiempo total incluido el procesamiento es de 12640 useg, menos los 470 useg del tiempo de lectura/escritura da 12170 useg.

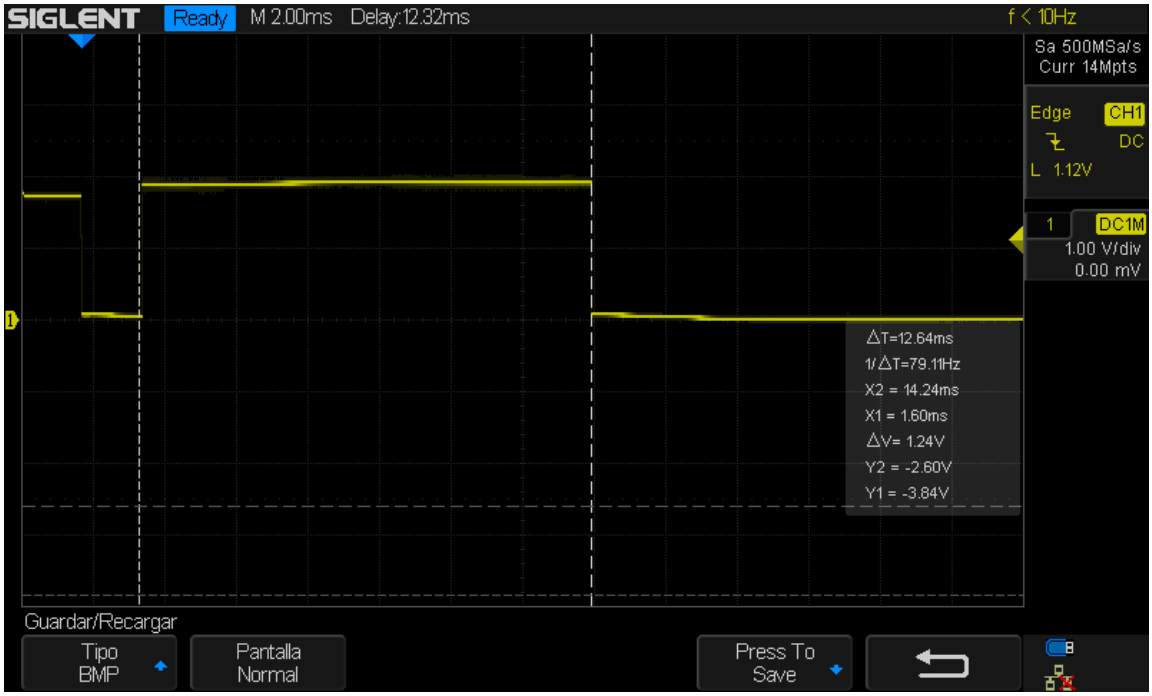

Figura 42.Tiempo de ejecución del filtro de sobel con cuatro lecturas, dos procesamientos y dos filas de salida

En la Tabla 5 se muestran los tiempos de procesamiento con dos procesadores respecto al uso de un solo procesador. Aquí se puede observar el beneficio obtenido al implementar un sistema multiprocesador que usa memoria compartida con múltiples puertos respecto a uno simple procesador, ya que al poseer prácticamente todo el código paralelizable la mejora es prácticamente el doble. El análisis que se puede hacer en relación a los tiempos de los distintos algoritmos es que a medida que la complejidad del algoritmo se incrementa, el tiempo de lectura/escritura de memoria compartida se vuelve despreciable respecto al de procesamiento.

Tabla 5. Tiempos de procesamiento de una fila usando uno y dos procesadores

Algoritmo	Un procesador (useg)	Dos procesadores (useg)
Filtro de mediana	3715	1870
Filtro de media	215	120
Filtro de sobel	6055	3040
Erosión/dilatación	3715	1870

5.2.3. Medición de tiempos de procesamiento implementando el algoritmo en un coprocesador

Luego de determinar los tiempos de procesamiento mediante dos procesadores, se realizaron pruebas de ejecución de los algoritmos con dos procesadores y la adición de los coprocesadores que se utilizan para acelerar la ejecución de los algoritmos.

Los tiempos obtenidos ejecutando el filtro de mediana con el coprocesador fueron de 0,9125ms, para una imagen de 128x128, sin tener en cuenta los tiempos de lectura y escritura de memoria compartida. Se debe tener en cuenta que para medir los tiempos de erosión y dilatación se utiliza el mismo algoritmo, solo que se toman los valores mínimos y máximos en lugar del pixel central, es por eso que los tiempos obtenidos en la Tabla 6 son iguales.

El filtro de media solo requiere la realización de nueve sumas y tres desplazamientos en VHDL, algo que tiene muy bajo costo computacional, pero a esto se le deben adicionar los tiempos de comunicación mediante el bus FSL entre el procesador y el coprocesador. El hecho de ser un algoritmo muy simple, hace que los tiempos de comunicación FSL comiencen a pesar en comparación con el tiempo de cálculo, es por eso que en este caso no se llegan a notar los beneficios del coprocesador.

En cuanto al algoritmo de sobel, la parte de la convolución se realizó de manera eficiente en VHDL, pero las operaciones finales de productos y raíz cuadrada se realizaron en el procesador.

Tabla 6. Tiempos de procesamiento de una fila usando dos procesadores con coprocesador

Algoritmo	Dos procesadores (useg)
Filtro de mediana	912,5
Filtro de media	130
Filtro de sobel	1960
Erosión/dilatación	912,5

5.3. Evaluación de Rendimiento

Para evaluar el rendimiento conseguido por los distintos sistemas que se probaron, se ejecutaron los algoritmos que se han descrito en el capítulo 3, midiendo el tiempo de ejecución sobre cada uno [29]. En cada caso evaluado se ejecutaron las aplicaciones primero sobre un procesador, y luego sobre dos procesadores con y sin coprocesador. Luego, para determinar el rendimiento se utilizaron las siguientes métricas:

- *SpeedUp:* representa la aceleración del sistema a la hora de ejecutar una aplicación respecto al sistema con sólo un procesador, esto es el cociente entre el tiempo que tarda el programa en ejecutarse sobre un solo procesador y el tiempo que tarda en n procesadores [29].

$$Speedup = \frac{T_1}{T_n}$$

- *Eficiencia relativa al número de procesadores:* representa el grado de aprovechamiento de los procesadores del sistema a la hora de ejecutar una aplicación paralela. Se define como el cociente entre el speedup y el número de procesadores.

$$E_P = \frac{S_p}{P}$$

- *Eficiencia relativa al área:* representa el grado de aprovechamiento de los recursos de la FPGA al ejecutar una aplicación paralela utilizando varios procesadores. Se define

como la división entre el speedup y el cociente de las áreas del sistema ocupadas por un procesador y por n procesadores.

$$C_a = \frac{A_1}{A_n} \quad E_a = \frac{S_n}{C_a}$$

5.3.1. Speedup

Para el cálculo del speedup se podría haber utilizado la ley formal de amdahl, que depende del porcentaje de código paralelizable y del número de procesadores, pero no es una buena métrica al momento de tener en cuenta los coprocesadores principalmente porque una parte del algoritmo se ejecuta en lenguaje C y otra en VHDL. En la Tabla 7 se muestran los valores de speedup para un sistema de dos procesadores con y sin coprocesador, entre los resultados se puede destacar que la mejora lograda para los algoritmos de mayor costo computacional es notable, esto se debe a que los tiempos consumidos en el movimiento de datos son despreciables respecto a los tiempos de cálculo. En cambio, en la ejecución del algoritmo de media no se logra ningún beneficio, debido a que los tiempos de cálculo son mucho menores y comparables con los tiempos de movimiento de datos.

Tabla 7. Cáculo de Speedup sin y con coprocesador

Algoritmo	Speedup sin coprocesador	Speedup con coprocesador
Filtro de mediana	1,987	4,07
Filtro de media	1,791	1,65
Filtro de sobel	1,991	3,089
Erosión/dilatación	1,987	4,07

Como se esperaba el gráfico de speedup de la figura 43 muestra un comportamiento bastante lineal, teniendo en cuenta que las pruebas realizadas incluyen uno, dos procesadores y dos procesadores más el coprocesador [27-28]. El único caso en donde no se observa mejora aparente, es el cálculo del algoritmo de medía que tiene características muy diferentes al resto de los algoritmos.

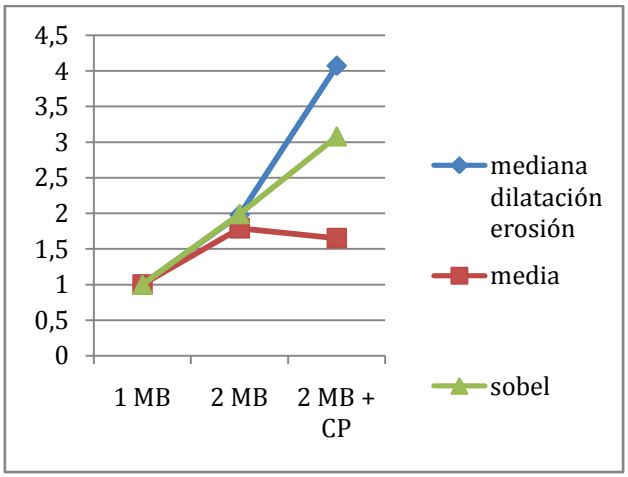

Figura 43. Speedup vs número de procesadores

5.3.2. Eficiencia relativa al número de procesadores

Respecto a la eficiencia relativa al número de procesadores, se observa en la Tabla 8 que la misma se incrementa a casi el doble al agregar a los procesadores un acelerador de hardware (coprocesador).

Tabla 8. Eficiencia relativa al número de procesadores

Algoritmo	$E_p$sin coprocesador	E_p con coprocesador
Filtro de mediana	0,9935	2,035
Filtro de media	0,8955	0,825
Filtro de sobel	0,955	1,5445
Erosión/dilatación	0,9935	2,035

5.3.3. Eficiencia relativa al área

Para el cálculo de la eficiencia relativa al área, primero se debe calcular el cociente de las áreas del sistema con un procesador y con dos procesadores. Para el cálculo del coeficiente C_a se tuvieron en cuenta los datos detallados en las tablas 1,2 y 3; de donde se desprende que dos procesadores ocupan el 34% de los bloques slice, un procesador el 15,78%, el filtro de ordenamiento en VHDL el 4,9%, la convolución el 2,23% y el promedio se considera despreciable. En conclusión, la mejora en la eficiencia relativa al área que se observa en la Tabla 9 se debe a que el área ocupada por los coprocesadores es despreciable respecto a la utilizada por cada procesador, sumado a la mejora lograda en el cálculo de speedup al usar los coprocesadores.

Tabla 9. Eficiencia relativa al área

Algoritmo	E_A	E_A con coprocesador
Filtro de mediana	0,9216	1,6507
Filtro de media	0,8307	0,7653
Filtro de sobel	0,9234	1,3468
Erosión/dilatación	0,9216	1,6507

5.4. Resultados del procesamiento

Entre los resultados se debe destacar la mejora que se logra (speedup) al realizar el procesamiento mediante el agregado de un bloque de HW coprocesador, pero también es importante el análisis de la imagen para verificar la efectividad del algoritmos y su correcta ejecución.

En cuanto al tipo de imágenes a procesar, este método se puede aplicar en la ejecución de algoritmos basados en operadores de ventana, sobre cualquier tipo de imágenes digitales. Se decidió aplicar a imágenes médicas principalmente por el gran caudal de datos de imágenes que deben procesar los equipos médicos como los de rayos x, PET y ecografías.

En la Figura 44 se puede observar el efecto de aplicar el filtro de media sobre imágenes médicas de 128x128 píxeles. En este caso se procesó una imagen radiográfica y una

tomográfica con ruido sal y pimienta, donde se observa que el efecto del ruido no es eliminado completamente, pero es minimizado, debido a que el promedio en una ventana no elimina valores extremos, sino que permite suavizar imágenes con irregularidades.

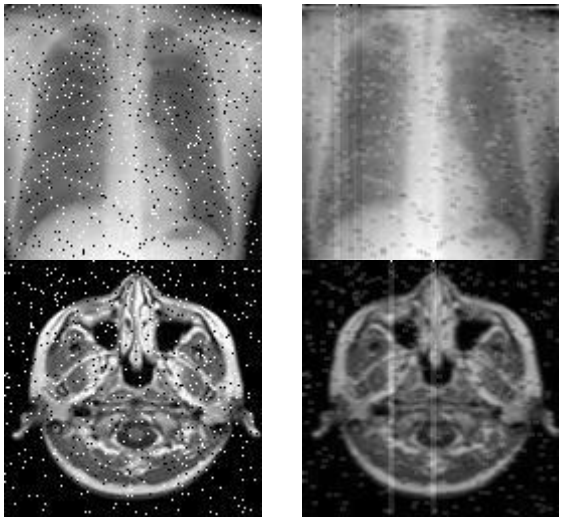

Figura 44. Procesamiento de una imagen radiográfica con el filtro de media

En la Figura 45 se observan los resultados de aplicar el filtro de mediana a las imágenes de 128x128. Aquí se observa la eficiencia de este método para eliminar el ruido de tipo sal y pimienta. Cabe destacar que el ruido fue generado de manera artificial, es por eso que en algunos casos donde hay varios píxeles vecinos con valores extremos, como el caso del negro en la imagen radiográfica, al realizar el ordenamiento puede suceder que el píxel central sea un valor extremo y por eso no se elimine completamente. Una posible solución podría ser realizar el procesamiento con una ventana de 5x5 en lugar de usar una de 3x3.

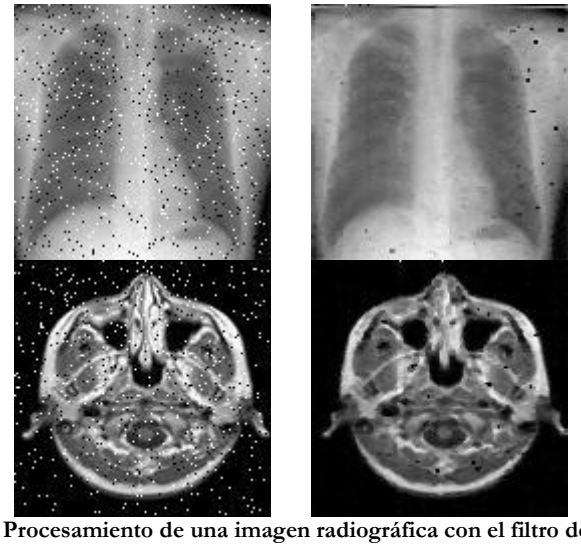

Figura 45. Procesamiento de una imagen radiográfica con el filtro de mediana

En la Figura 46 se aplicó el filtro de sobel para obtener los bordes de los objetos de la imagen que son un conjunto de embriones [26]. En este caso lo que interesa es demarcar los bordes de los objetos de un color diferente para poder individualizarlos y determinar su tamaño en función de la cantidad de píxeles o sus formas.

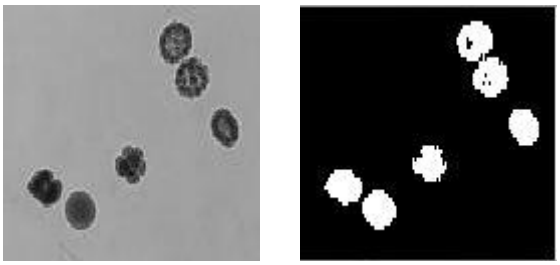

Figura 46. Procesamiento de una imagen de embriones con el filtro de sobel

Por último, en la Figura 47 se observa el efecto del algoritmo de dilatación, que es muy utilizado luego de aplicar algún algoritmo que reduce el tamaño de los objetos, por ejemplo para separar dos objetos pegados. El objetivo de este algoritmo es devolver el tamaño original a cada objeto para poder determinar su tamaño correctamente.

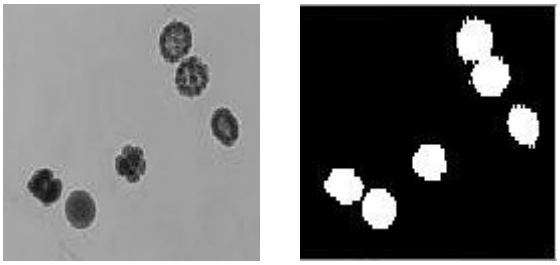

Figura 47. Procesamiento de una imagen de embriones con el método de dilatación

6. Conclusiones

En este trabajo se muestra la potencialidad de las plataformas FPGAs para implementar sistemas multi-core y explotar las bondades de la lógica programable mediante la implementación de un coprocesador en lenguaje de descripción de hardware en VHDL. La flexibilidad de estos sistemas permite evaluar diferentes diseños en la búsqueda de eficiencia al momento de procesar algoritmos de imágenes con características específicas (basados en operadores de ventana) [25-26]. Este tipo de plataformas reconfigurables es ideal para la investigación de sistemas que combinan procesamiento paralelo con la concurrencia que provee el lenguaje VHDL.

Mediante los resultados obtenidos se demostró que el sistema diseñado permite procesar de manera eficiente los algoritmos de procesamiento basados en operadores de ventana, aprovechando las características paralelizables que proveen los datos de este tipo de algoritmos. Por la característica de los datos a procesar, la técnica de procesamiento paralelo utilizada necesariamente debe ser de tipo SIMD, debido a que se ejecuta la misma porción de código de los algoritmos en cada procesador pero sobre diferentes sectores de la imagen (diferentes datos). Se debe tener en cuenta que por las características de estos algoritmos, los bloques de píxeles accedidos por cada procesador deben estar parcialmente solapados en la mitad de la imagen, (para imagen de 128 filas, se solapa sobre la fila 64), pero por las características del sistema, el hecho de tener las filas centrales en común a cada procesador no afecta a la eficiencia durante el procesamiento.

Los tiempos obtenidos en la ejecución de los algoritmos usando uno y dos procesadores fueron muy satisfactorios debido a que cada procesador accede a datos ubicados en diferentes partes de la memoria mediante puertos independientes. Gracias a esto se logró un speedup muy cercano a dos para todos los algoritmos [28]. Se debe destacar que el hecho de poseer una memoria compartida de múltiples puertos posibilitó el acceso simultáneo a los datos por parte de los procesadores.

La incorporación de los coprocesadores para la ejecución de la parte principal de los algoritmos, permitió reducir en gran medida los tiempos de procesamiento en los algoritmos de mediana, erosión, dilatación y sobel [25-27]. Esto se debe principalmente a que estos algoritmos requieren para su ejecución gran cantidad de instrucciones y operaciones complejas. La mejora en el desempeño del sistema se vio reflejada en el incremento del speedup a valores cercanos a cuatro. Se debe destacar que la ejecución del algoritmo de media mediante el coprocesador no mostró ningún beneficio debido a que es un algoritmo con pocas y simples instrucciones cuyo tiempo de ejecución es comparable con los tiempos de movimientos de datos.

Respecto a los recursos consumidos, se puede resaltar que es despreciable el incremento de los recursos utilizados por el coprocesador respecto a los necesarios para la implementación de cada microblaze. Esto último, se ve reflejado en el cálculo de eficiencia relativa al área, en donde la diferencia en el cálculo del cociente entre áreas es mínima comparada con la mejora en el speedup [25-28].

Si bien las tecnologías utilizadas están quedando en desuso, se debe destacar que las características del sistema implementado permiten una total portabilidad hacia las nuevas tecnologías, tal como se explica en el Apéndice B secciones B.1.2 y B.1.3.

Como tareas a futuro se proponen:

- Implementar el sistemas con 3, 4 y 8 procesadores para verificar las estimaciones teóricas realizadas
- Aplicar principios de HPRC (cómputo de altas prestaciones reconfigurable) para mejorar el desempeño del sistema
- Implementar una interfaz de comunicaciones más eficiente con el host(Ethernet o USB)
- Ampliar las técnicas de procesamiento a otros algoritmos de procesamiento de imágenes con características diferentes mediante técnicas de sistemas dinámicamente reconfigurables

Bibliografía

[1] John L. Semmlow: *Biosignal and Biomedical Image Processing*, Marcel Dekker, New Jersey, 2004.

[2] Andraka Consulting Group, Inc,:*Digital Signal Processing for FPGAs*, Seminar Notes, 1999.

[3] Branislav Kisacanin, Shuvra S. Bhattacharyya, Sek Chai: *Embedded Computer Vision*, Ed. Springer, Verlag London, 2009.

[4] William K. Pratt: *Digital Image Processing*, Wiley, Los Altos California, 2004.

[5] Dougherty, E.: *An Introduction to Morphological Image Processing*, SPIE, Bellingham, WA, 1992.

[6] PITAS, I. y VENETSANOPOULOS, A. N., *Nonlinear digital Filters: principies and applications*, Kluwer Academic Publishers, Norwell, Massachusetts, 1991.

[7] Rafael C. González, Richard E. Woods, Steven L. Eddins, *Digital Image Processing using Matlab*, Prentice-Hall, New Jersey, 2004.

[8] HUSSAIN, Z., *Digital image processing: practical applications of parallel processing techniques*, Ellis Horwood, Chichester-lnglaterra, 1991.

[9] M.J, *Flynn, Computer organizations and their effectiveness*. 1972, IEEE Transactions on Computers, pp. 948-960.

[10] J.L. Hennesy, D.A Patterson. *Computer Architecture A Quantitative Approach*.Cuarta edición. Morgan Kauffman, 2007.

[11] D.E. Culler, J.P. Singh. *Parallel Computer Architecture a Hardware/Software Approach*.Morgan Kauffman, 1999.

[12] A. Grama, A. Gupta , G. Karypis , V. Kumar. *Introduction to Parallel Computing*.Segunda edición. Addison Wesley, 2003.

[13] Johnson, E. E. *Completing an MIMD multiprocessor taxonomy*. 1988, ACM SIGARCH Computer Architecture News, Vol. 16, pp. 44-47.

[14] Parhami, Behrooz, *Introduction to Parallel Processing*. s.l. : Springer US, 2006. 1567-7974.

[15] Jesman R., Martinez Vallina F., Saniie J., *MicroBlaze TutorialCreating a Simple Embedded SystemandAdding Custom Peripherals UsingXilinx EDK Software Tools,* Illinois Institute of technology, 2006.

[16] *MicroBlaze Processor Reference Guide – Embedded Development kit EDK 13.2* , Xilinx, Junio de 2011

[17] Vasanth Asokan, *Designing Multiprocessor Systems in Platform Studio,* Xilinx White Paper WP262, Noviembre de 2007.

[18] Vasanth Asokan, *Dual Processor Reference Design Suite*, Aplication Note: embedded Processing, Octubre de 2008.

[19] Moreno Franco Olmo Alonso*, Diseño y Programación de Sistemas Embebidos Con El NúcleoMicroblaze* ,Eae Editorial Academia Española, Marzo 2012.

[20] PeterJ. Ashenden, *The Designer's Guide to VHDL 2nd Edition*, Editorial Morgan Kaufman, 2002

[21] Pong P. Chu, *FPGA Prototyping by VHDL Examples*, Xilinx Spartan 3 Version, John Wiley Cleveland

[22] Nelson, A. *Further Study of Image Processing Techniques on FPGA Hardware*, *Independent Study Paper*, May 1999.

[23] JORGE OSIO; JOSÉ RAPALLINI; ANTONIO ADRIÁN QUIJANO; JESÚS OCAMPO. *Implementación de un Algoritmo para procesamiento de imágenes en una FPGA*. Libro de memorias del Primer Congreso de Microelectrónica Aplicada. 2010. Artículo Completo. Congreso de microelectrónica Aplicada UEA 2010. Universidad Nacional de la Matanza; Universidad

[24] *Synthesis and Simulation Design Guide*, Xilinx

[25] Osio J., Aróztegui W, Rapallini J., Quijano A., Ocampo J., *Desarrollo de Algoritmos de Procesamiento de Imágenes Basados en Operadores de Ventana sobre una FPGA*. XVII Congreso de Iberchip. Bogotá: Universidad Nacional de Colombia. 2011 vol.1 n°1. p156 - 161. issn 2177-1286.

[26] JORGE R. OSIO, WALTER ARÓZTEGUI, JOSÉ A. RAPALLINI, ANTONIO QUIJANO. *Procesamiento de Imágenes Médicas Sobre una FPGA para la detección de bordes*. Argentina. Rosario. 2012. Libro. Artículo Completo. Tercer Congreso de Microelectrónica Aplicada. Universidad Nacional de Rosario

[27] OSIO J., MONTEZANTI D., MORALES M. *Análisis de Eficiencia en Sistemas Paralelos*. Argentina. Ushuaia. 2014. Libro. Workshop. XVI WORKSHOP DE INVESTIGADORES EN CIENCIA DE LA COMPUTACIÓN. Universidad Nacional de Tierra del Fuego, Antártida e Islas del Atlántico Sur.

[28] OSIO J. R, AROZTEGUI W., QUIJANO A. A., RAPALLINI J. A. *Determinación de eficiencia en la ejecución de algoritmos de procesamiento de imágenes con múltiples procesadores en FPGA*. Argentina. Santa Fe. 2019. Libro. Congreso. Congreso Argentino de Sistemas Embebidos. Universidad Nacional de Rosario, Santa Fe, Argentina. ISBN 978-987-46297-6-0

[29] William Stallings, *Organización y Arquitectura de Computadores*, Editorial Prentice Hall, 8va edición, 2010.

[30] James O. Hamlen Michael D. Furman, *Rapid Prototyping of Digital Systems*, Kluwer Academic Publishers, 2000.

[31] John F. Wakerly ,*Diseño Digital Principios y Prácticas*, Editorial Prentice Hall, 3ra Edición, 2001.

A. Características básicas de las FPGAs

A.1. Introducción

La arquitectura de una FPGA consiste en arreglos de bloques lógicos que se comunican entre sí a través de canales de conexión verticales y horizontales. La principal diferencia entre las FPGA y CPLDs es que en general los bloques lógicos de las FPGA no implementan la lógica usando compuertas sino generadores de funciones [30].

Típicamente la arquitectura general de una FPGA contiene cinco elementos principales:

- *Bloque de entrada-salida* (IOB o Input-Output Block): estos bloques proveen la interfaz entre los "pines" del integrado y la lógica interna.
- *Bloque lógico configurable* (CLB o Configurable Logic Block): Estos son los bloques básicos que se utilizarán en la implementación de un circuito digital.
- *Bloque de distribución y compensación de reloj* (DLL o Delay Locked Loop): Estos bloques controlan los dominios de reloj dentro del integrado y compensan por retardos que puedan existir entre el reloj externo y el interno.
- *Bloque de memoria* (BLOCK RAM): Estos bloques son memorias dedicadas integradas dentro de la lógica programable.
- *Estructura de interconexión:* Es una estructura versátil y multi-nivel de interconexión entre los otros componentes de la FPGA.

Además de los bloques lógicos básicos, que ocupan la mayor parte de las FPGAs, (CLBs, Logic Array, Logic Tile, etc.), estos dispositivos constan de bloques que permiten realizar ciertas tareas específicas. No todos los bloques se encuentran en todas las FPGA, aunque casi todas tienen la estructura general presentada anteriormente.

A.2. Bloques lógicos programables

Todas las FPGA tienen algún tipo de bloque lógico programable. Esta es la esencia de la FPGA y permite implementar las diferentes funciones lógicas. En la Figura 48 se muestra un esquema del bloque lógico programable (CLB) de una FPGA de la familia Spartan 3.

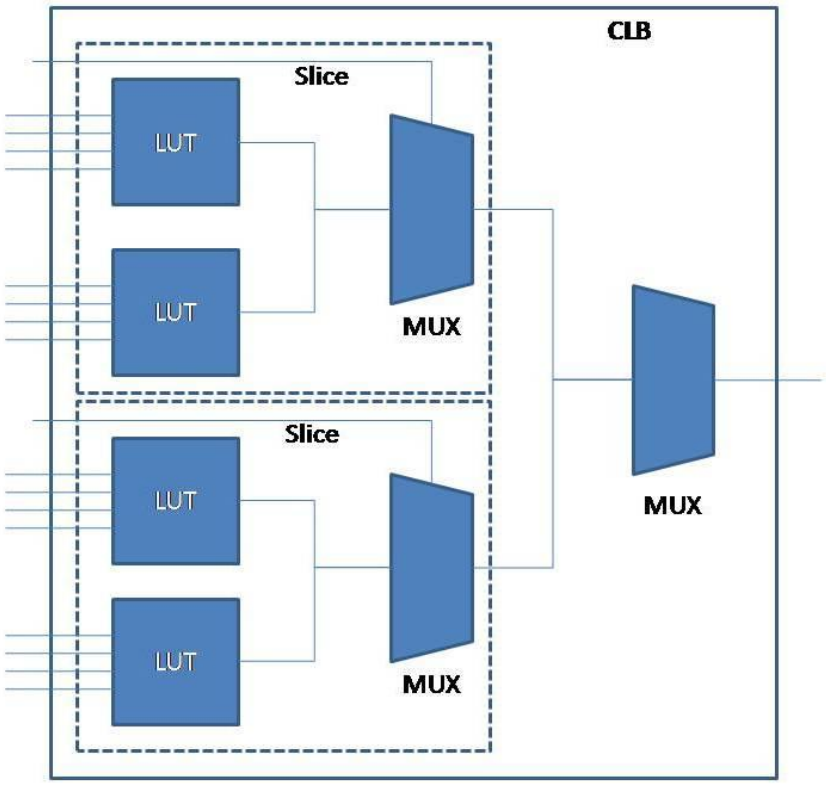

Figura 48. Esquema del bloque lógico configurable de una FPGA Spartan de Xilinx

Cada CLB está compuesto por dos bloques iguales denominados "slices". Cada "slice" contiene dos generadores de funciones y un multiplexor MUX1. El multiplexor combina los resultados de los generadores de funciones dentro de cada "slice" del CLB.

Las dos "slices" están unidas por un multiplexor MUX2, que puede seleccionar la salida de una u otra "slice" hacia la salida del CLB. Esto permite implementar cualquier función de 6 entradas, un multiplexor de 8:1 o determinadas funciones lógicas de hasta 19 entradas. Además, de poder implementarse lógica combinacional, cada "slice" contiene recursos para implementar circuitos secuenciales y operaciones aritméticas.

Los generadores de funciones del slice están compuestos por una tabla de entrada-salida (LUT o tabla de Look-Up) de cuatro entradas y una salida. Estas tablas pueden implementar cualquier función lógica de cuatro entradas y una salida, así como implementar memorias distribuidas de 16 x 1 bit. Las salidas de las LUT pasan a los bloques de control que contienen lógica que permite optimizar funciones aritméticas. Los elementos que permiten implementar lógica secuencial son los elementos de almacenamiento que están sobre las salidas del slice y pueden configurarse como flip-flops D con reloj o como latches controlados por nivel.

A.3. Bloque de Entrada/Salida

Las interfaces de entrada-salida son otros de los componentes particulares que tienen las FPGAs. La familia de FPGAs Spartan III de Xilinx, por ejemplo, divide las entradas/salidas del integrado en bancos que se pueden configurar para tener una interfaz con lógica de diferentes niveles de tensión de manera independiente. Los bancos se configuran aplicando tensiones de alimentación a los pines denominados VccO y VREF. Al utilizar varios valores de VccO para

los distintos bancos se podrá tener un sistema con interfaz a diferentes familias lógicas dentro de la misma FPGA. Las entradas de reloj están asociadas a cada banco de entrada-salida para permitir que haya diferentes dominios de reloj con interfaces eléctricas diferentes. La Figura 49 muestra un esquema de estos bancos de entrada-salida.

Figura 49.Distribución de los bancos de entrada/salida en una FPGA Spartan3E de Xilinx

La siguiente tabla muestra el valor de VccO para las diferentes interfaces lógicas.

Tabla 10. Tensión VccO para diferentes interfaces lógicas de la familia Spartan de Xilinx.

Valor de VccO	Lógica de interfaz
3.3V	PCI, LVTTL, SSTL3 I, SSTL3 II, CTT, AGP, LVPECL, GTL, GTL+
2.5V	SSTL2 I, SSTL2 II, LVCMOS2, LVDS, Bus LVDS, GTL, GTL+
1.8V	LVCMOS18, GTL, GTL+
1.5V	HSTL I, HSTL III, HSTL IV, GTL, GTL+

Además de la tensión VccO, para varios de las interfaces lógicas debe configurarse la tensión de referencia VREF y agregarse resistencias de terminación, sobre el circuito impreso, en las entradas-salidas de la FPGA. Cada bloque de entrada-salida tiene una resistencia de "pull-up" y "pull-down" configurables que permiten fijar el valor lógico mediante programación.

A.4. Bloque de control de reloj

El sistema de control del reloj consiste en bloques de control integrados a la red de distribución de reloj. Esta red de distribución en las FPGAs asegura retardos parejos a todos los bloques lógicos de la FPGA. Cada fabricante utiliza una arquitectura diferente para el control y distribución de la señal de reloj. La Figura 50 muestra el esquema de distribución de reloj para la familia Spartan de Xilinx.

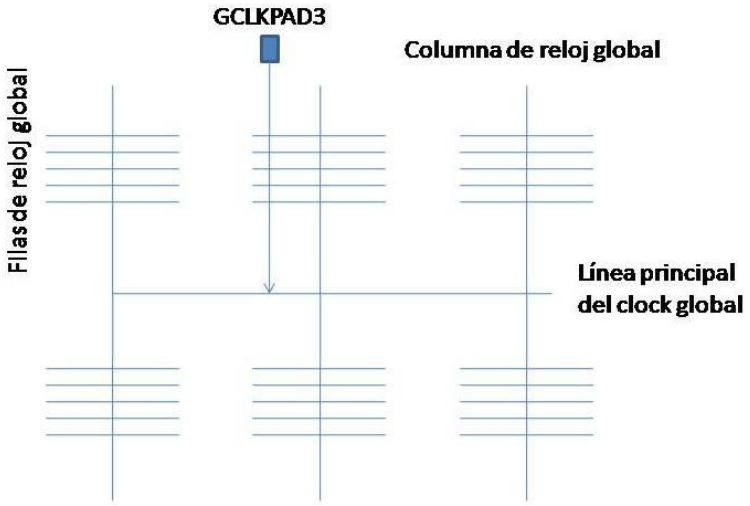

Figura 50. Red global de distribución de reloj en la FPGA Spartan 3

A continuación se presenta un diagrama en bloques del control de reloj. La familia Spartan de Xilinx tiene bloques específicos para control de reloj denominados DLL (Delay Locked Loop). La Figura 51 muestra un esquema básico de la estructura de un DLL, estos bloques sincronizan el reloj interno con el reloj externo del sistema, controlan el desplazamiento de fase entre los relojes, sincronizan los diferentes dominios de reloj y aseguran un retardo de distribución similar para la lógica interna.

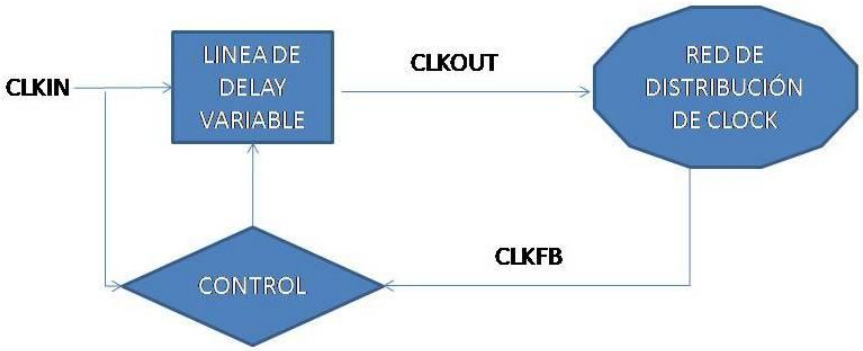

Figura 51. Esquema básico del bloque de control de reloj de la FPGA Spartan

A.5. Memoria

Varias familias de FPGA contienen bloques de memoria embebida integrados con la lógica programable. Estos bloques básicos de memoria pueden utilizarse en diferentes configuraciones para generar RAMs y ROMs de diferentes tamaños. Además de las memorias

embebidas, las FPGAs basadas en memoria SRAM pueden usar las tablas LUT de los bloques lógicos como memoria. La celda de memoria se denomina BLOCK RAM y es una memoria de puerto dual (dual-port), que puede leerse y escribirse al mismo tiempo [31]. En la siguiente tabla se muestran las opciones de configuración de los bloques de memoria. Las interfaces de dirección y datos (ADDRA, ADDRB, DIA, DOB, DOA) se pueden configurar para diferentes tamaños de memoria.

Tabla 11. Posibles configuraciones de las celdas de BLOCK RAM de la familia Spartan

Ancho de la palabra de datos (bits)	**Profundidad de la memoria**	**Bus de direcciones**	**Bus de datos**
1	4096	ADDR<11:0>	DATA<0>
2	2048	ADDR<10:0>	DATA<1:0>
4	1024	ADDR<9:0>	DATA<3:0>
8	512	ADDR<8:0>	DATA<7:0>
16	256	ADDR<7:0>	DATA<15:0>

A.6. Bloque de procesamiento de señal

Varias FPGAs contienen bloques específicos que optimizan en hardware ciertas funciones especiales, por ejemplo, contienen uno o más módulos de procesamiento de señal entre los bloques de lógica programable de propósito general. Estos bloques permiten desarrollar ciertas funciones específicas, típicas de las aplicaciones de procesamiento de señal como es hacer productos y desplazamientos. Estas tareas requerirían de muchos recursos y ciclos de reloj si se implementaran utilizando lógica de propósito general.

A.7. CPUs Embebidas

Hay familias de FPGAs que contienen una CPU Power PCy lógica de interconexión embebida dentro de lógica programable, esto permite utilizar toda la potencia de una CPU integrada con la flexibilidad de los periféricos diseñados mediante lógica programable. Los bloques específicos integrados en el silicio de las FPGAs se denominan "hardcores".

B. Particularidades de la implementación

B.1. Tecnologías de dispositivos FPGAs

B.1.1. Herramientas de diseño utilizadas

En la Figura 5 se muestra el kit de desarrollo utilizado para la implementación del sistema. Este, es un kit de la empresa Digilent y tiene las siguientes características:

- FPGA Virtex-5 LX50T - BGA de 1136 pines
- DDR2 SODIMM de 256Mbyte con datos de 64 bits
- StrataFlash de 32Mbyte para configuración y almacenamiento de datos de usuario
- Oscilador de 100 MHz y generador de reloj de hasta 400 MHz
- Interfaz Ethernet PHY de 10/100/1000 y puerto serie RS-232
- Puertos USB2 para aplicaciones de programación, transferencia de datos y hosting
- Puertos UART
- Video HDMI con resolución de hasta 1600x1200 y color de 24 bits
- LEDs, pulsadores, interruptor, swicthes y display LCD de 16x2

Figura 52. Kit de desarrollo Digilent con virtex V

Las herramientas utilizadas para el diseño y programación del sistema fueron el ISE Design Suite para programación, síntesis e implementación del código VHDL (ver Figura 53); el EDK para

el diseño y configuración del Hardware de procesamiento (ver Figura 54) y el SDK para el diseño y programación del Software embebido de los procesadores MicroBlaze, que se muestra en la Figura 55.

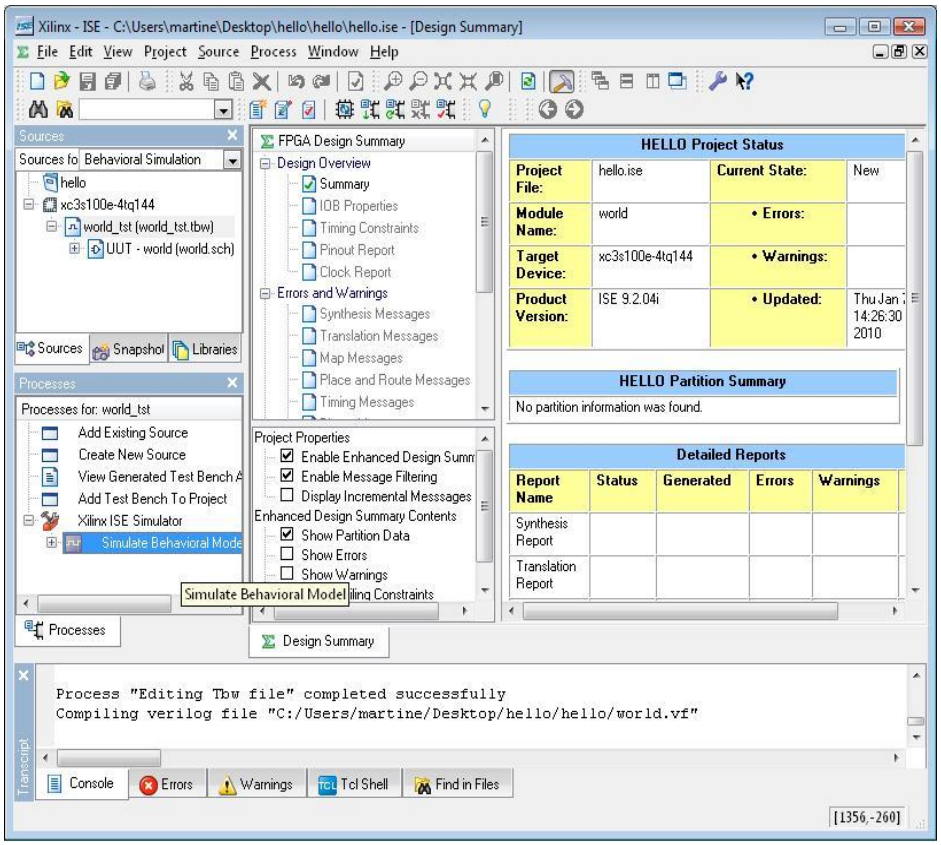

Figura 53. Software ISE Design Suite

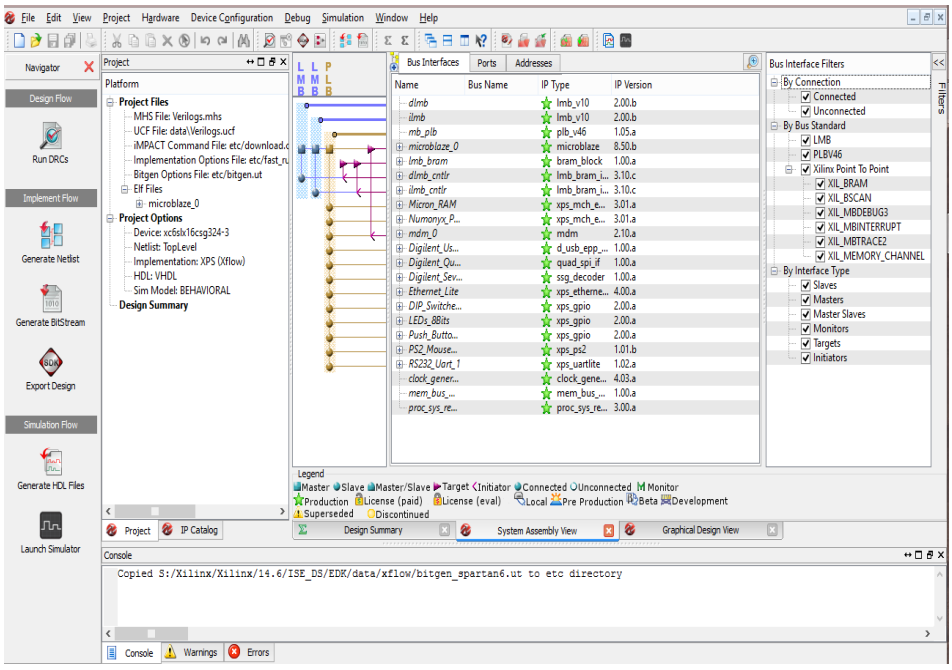

Figura 54. Software SDK

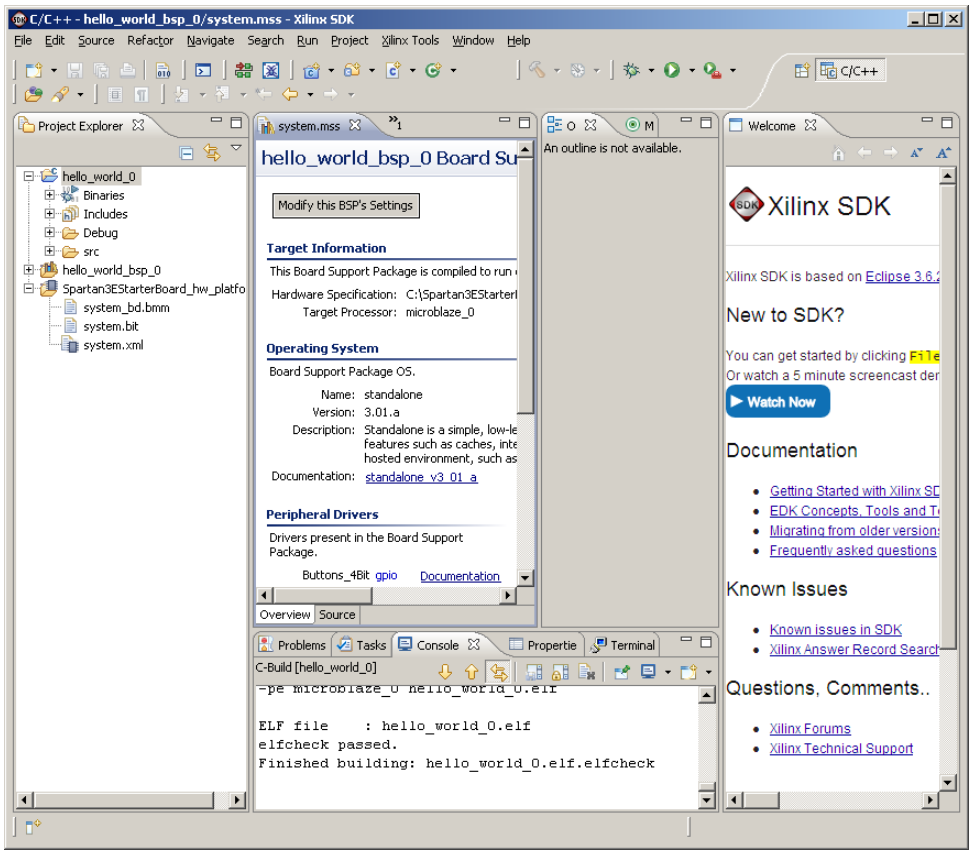

Figura 55. Software EDK

B.1.2. Dispositivos FPGAs actuales

Las tecnologías de FPGAs actuales están basadas en la Arquitectura SoC Zynq-7000 y tienen las siguientes características:

- PS: Sistema de Procesamiento
  - Basado en el procesador ARM Cortex-A9
  - Múltiples periféricos
- PL: Lógica Programable
  - Usa las misma lógica programable de la serie 7
    - Dispositivos basados en ARTIX: Z-7010, Z-7015 y Z-7020
    - Dispositivos basados en KINTEX: Z-7030, Z-7035, Z-7045 y Z-7100

Para el diseño e implementación de sistemas digitales con estas tecnologías se utiliza el Software de Diseño Integrado VIVADO Design Suite, que provee herramientas muy potentes para la programación de procesadores, la configuración y programación de hardware. Las herramientas que provee VIVADO son:

- Editor amigable para VHDL y Verilog
- Herramienta integradora de IP
- Catalogo de IP (xilinx y third party)
- Herramientas para implementación y síntesis
- Generador de bitstream

- Herramienta de simulación XSim

B.1.3. Portabilidad del sistema a las plataformas actuales

Teniendo en cuenta que el sistema se diseñó con un dispositivo prácticamente es desuso, es importante analizar la portabilidad del sistema diseñado hacia las plataformas actuales. A continuación se describe el soporte que da VIVADO para la implementación del Sistema sobre las Arquitecturas SoC Zynq-7000:

- Sistema Multiprocesador
  - MicroBlaze (soft-processor)
  - Ideal para sistemas multiprocesador
- Buses
  - MicroBlaze contempla FSL, PLB y MLB
- Etapa Coprocesador
  - Vivado provee herramientas para optimización de código en lenguaje de descripción de HW VHDL

B.2. Pasos de diseño de un sistema multiprocesador

1. Cree un diseño de procesador único implementado sobre una placa de desarrollo particular, con todos los periféricos necesarios, memoria local y externa usando el asistente de Base System Builder.

2. Agregue el segundo procesador mediante arrastrar y soltar en la ventana de diseño y conéctelo a un nuevo bus de sistema PLBv46.

3. Conecte la interfaz de depuración del procesador al periférico de depuración (JTAGPPC o MDM). Configure el periférico de depuración, si es necesario, para manejar el segundo procesador también.

4. Proporcione memoria local (para mantener el código de arranque) al procesador recién agregado. Esta es la memoria ILMB y DLMB para el procesador MicroBlaze o IOCM y DOCM para el procesador PowerPC. Los buses de memoria local se deben agregar primero, seguidos por una conexión al procesador y una conexión al controlador de memoria. Alternativamente, esta memoria de arranque puede estar en el bus PLBv46.

5. Agregue nuevos puertos y conecte la memoria externa mediante el controlador MPMC al segundo procesador. Conecte enlaces de caché si es necesario.

6. Agregue los componentes de comunicación interprocesador (Mutex y buzón) y configúrelos.

7. Agregue memoria local compartida conectándola a través de DLMB para el procesador MicroBlaze y DOCM para el procesador PowerPC. El bus PLBv46 también se puede usar para interactuar con la memoria BRAM local compartida.

8. Agregue periféricos de interfaz, si es necesario.

9. Agregue otros periféricos adicionales, como el temporizador o el controlador de interrupción, que son necesarios para ejecutar aplicaciones de software razonables.

10. Vaya a la vista de direcciones en el panel de ensamblaje del sistema y configure un rango de direcciones válido para todos los periféricos y memorias que se agregaron anteriormente.

11. Verifique las conexiones de clock y reset a todos los periféricos agregados desde los cores clock_generator y proc_sys_reset.

B.3. Código de transmisión y visualización de imágenes en el PC-Host

El código en matlab para la lectura de la imagen, la transmisión, la recepción y el almacenamiento se implementó como se muestra a continuación:

```
I= imread('xray128x128.jpg');    % lectura de la imagen
imagentemp=I;
[xlenght ylenght]=size(I);

PS=serial('COM1');                %configuración del Puerto serie
set(PS,'Baudrate',9600);
set(PS,'StopBits',1);
set(PS,'DataBits',8);
set(PS,'Parity','none');
set(PS,'InputBufferSize' ,xlenght);
set(PS,'TimeOut',200);
fopen(PS);                         % apertura del Puerto serie

L=[ ];
tic;                               % inicialización del temporizador
for m=1: xlenght                   %transmisión de la imagen
for n=1:ylenght
fwrite(PS,I(m,n),'uint8');
end
end
for j=1: xlenght% recepción de la imagen procesada
for i=1:ylenght
imagentemp(j,i)=[L fread(PS,1,'uint8')];
end
end
tiempo=toc;                        % tiempo de transmisión y procesamiento total
fclose(PS);                        % cierre del puerto serie
delete(PS);
clear PS;
 INSTRFIND                         %se verifica el cierre del puerto com
imshow(imagentemp);                % visualización y almacenamiento de la imagen procesada
imwrite(imagentemp,'media128x128.jpg');
```

C. Códigos C de los procesadores MB0 y MB1

Para la utilización de las bibliotecas de funciones de los procesadores Microblaze y la inicialización de los periféricos se deben agregar las siguientes líneas de código:

```
#include <stdio.h>
#include <math.h>
#include "xparameters.h"
#include "xil_cache.h"
#include "xbasic_types.h"
#include "gpio_header.h"
#include "xuartns550_l.h"
#include "xuartns550.h"
#include "xbasic_types.h"
#include "xbram.h"
#include "bram_header.h"
#include "mbox_header.h"
#include "uartlite_header.h"
#include "mutex_header.h"
#include "xtmrctr.h"
#include "tmrctr_header.h"
#include "platform.h"
#include "xil_testmem.h"
#include "xil_types.h"
#include "xstatus.h"
#include "memory_config.h"
#include "xgpio.h"

#define sizeimage            128
#define sizewindow 3
#define lectura 0
#define escritura 1
#define recibe 0
#define envia 1

void print(char *str);
void putnum(unsigned int num);

#define LED_CHANNEL 1
#define GPIO_BITWIDTH       16        /* Este es el ancho del GPIO */
#define printf xil_printf                   /* permite transmitir cadenas de caracteres */

/************************** Definición de variables **************************/
/* * lo que sigue se declara globalmente para que sea fácil de acceder por el debugger */
XGpio GpioOutput; /* La instancia del driver para configurar las GPIO como salida */
XGpio GpioInput;  /* La instancia del driver para configurar las GPIO como entrada*/
/*****************************************************************************/

//Función de configuración de los leds de la placa como salida para la medición de tiempo

int GpioOutputLed(u16 DeviceId, u32 GpioWidth)
  {
        int Status;
```

```
        Status = XGpio_Initialize(&GpioOutput, DeviceId);
        if (Status != XST_SUCCESS)
          {
                return XST_FAILURE;
           }
        XGpio_SetDataDirection(&GpioOutput, LED_CHANNEL, 0x0);
        XGpio_DiscreteWrite(&GpioOutput, LED_CHANNEL, 0x0);
        return XST_SUCCESS;
    }

//Función de lectura y escritura en memoria compartida

int Xil_Mem8(u8 *Addr, u32 Words, u8 Pixel[ ], u8 Flag)
    {
        u32 I;
        Xil_AssertNonvoid(Words != 0);
        if(Flag==1) //si flag es 1 se escribe en la memoria
            {
                for (I = 0L; I < Words; I++)
               {
                   Addr[I] = Pixel[I];   /* escribe en la memoria */
                 }
            }
        if (Flag==0) //si flag es cero se lee la memoria
            {
                  for (I = 0L; I < Words; I++)
                {       /* lee el contenido de la memoria */
                        Pixel[I] = Addr[I];
                }
            }
        return 0;
    }

/**********************INICIO DEL PROGRAMA PRICIPAL**************************/
//en esta sección se definen las variables y se inicializan los periféricos
/*****************************************************************************/

int main( )
        {
         int pixeles_input1 [128]={0}; //primer fila necesaria para la ventana valida
          int pixeles_input2 [128]={0};//segunda fila necesaria para la ventana valida
          int pixeles_input3 [128]={0};//tercera fila necesaria para la ventana valida
          int pixeles_output [128]={0}; //arreglo para almacenamiento temporal del resultado del
                                        //procesamiento
         u8 pixeles_output1 [128]={0};
         int pixeles_outputx [128]={0}; //arreglo temporal para sobel
         int pixeles_outputy [128]={0}; //arreglo temporal para sobel
         int j=0,i=0,k=0;
        int m=0,n=0,aux;
         int temp[9];
        u32 LedBit=0;
         u8 max=255, tmax=100; //tmax es el umbral para binarizar
         int kernely[3][3]={{-1, 0, 1 },{ -2,0, 2 },{ -1,0, 1}}; //kernel y para sobel
         int kernelx[3][3]={{-1,-2, -1} , {0, 0 ,0} , {1 ,2,1}}; //kernel x para sobel
         unsigned basei= 0x88000000;     //dirección base de datos de entrada
```

```
unsigned bases= 0x88004000;    //dirección base de datos de salida

 /* Inicializa RS232_Uart_0 – Setea el baudrate y el número de bits de stop */
   XUartNs550_SetBaud(XPAR_RS232_UART_0_BASEADDR,
   XPAR_XUARTNS550_CLOCK_HZ, 9600);
   XUartNs550_SetLineControlReg(XPAR_RS232_UART_0_BASEADDR,
   XUN_LCR_8_DATA_BITS);

 Xil_ICacheEnable();
 Xil_DCacheEnable();
```

C.1. Código implementado en Microblaze 0

C.1.1. Código de recepción, almacenamiento, lectura de memoria compartida y transmisión de píxeles

Recepción desde el PC-Host y almacenamiento en memoria compartida

```
 /*************************Recepción de datos desde el PC-host*****************************/
//En una instancia inicial, se recibe la imagen completa por puerto serie y se almacena en memoria
//compartida
/*****************************************************************************************/

 for (k=0; k<sizeimage; k++)
    {
        //Se recibe una fila desde el PC-Host
        for(i=0; i<sizeimage; i++)
         pixeles_input1[i]=XUartNs550_RecvByte(STDOUT_BASEADDR);
         //se almacena la fila completa en memoria compartida
       Xil_Mem8((u8*) basei+i*128, sizeimage, pixeles_input1,escritura);
     }
```

Código de Lectura de filas de la imagen para el posterior procesamiento

```
/************************ Lectura de memoria compartida *********************************/
// La lectura junto con la ejecución del algoritmo se realiza dentro de un bucle que lee la mitad de
//la imagen
/*****************************************************************************************/
  if(k==0)     //si se deben procesar las primeras ventanas, es necesario leer las tres
                        //primeras filas de la segunda mitad
          {
         Xil_Mem8((u8*) basei+8192, sizeimage, pixeles_input1, lectura);
          Xil_Mem8((u8*) basei+8192+128, sizeimage, pixeles_input2, lectura);
          Xil_Mem8((u8*) basei+8192+256, sizeimage, pixeles_input3, lectura);
         }
  if(k>0) //a partir de la segunda ventana se requiere leer una fila para procesar nuevas
         //ventanas
         {
            if(j==1)
               { Xil_Mem8((u8*) basei+8192 +(i+3)*128, sizeimage, pixeles_input1,lectura);}
            else if (j==2)
               {  Xil_Mem8((u8*) basei+8192 +(i+4)*128, sizeimage, pixeles_input2,lectura);}
            else
```

```
            {Xil_Mem8((u8*) basei+8192 +(i+5)*128, sizeimage, pixeles_input3,lectura);
            j=1;
              }
              j++;
        }
```

Código de Transmisión del resultado al PC-Host

```
/*****************************************************************************/
//Este código envía el resultado a la pc
/*****************************************************************************/

 for(k=0;k<sizeimage;k++)
   {//se lee de a una fila desde memoria compartida
       Xil_Mem8((u8*) bases+k*128, sizeimage, pixeles_output, lectura);
      //se transmite por puerto serie hacia el PC-Host
       for(i=0;i<sizeimage; i++)
           XUartNs550_SendByte(STDOUT_BASEADDR, pixeles_output[i]);
    } //fin de bucle de transmission hacia PC-Host
```

C.1.2. Código de los algoritmos de procesamiento

```
 //inicialización del pin utilizado para medir el tiempo de procesamiento
   {
         u32 status;
         //se inicializa pines para medir tiempo
         status = GpioOutputLed(XPAR_LEDS_8BIT_DEVICE_ID,8);

        //se pone en alto el pin del led 0 para medir tiempo desde el osciloscopio
        XGpio_DiscreteWrite(&GpioOutput, LED_CHANNEL,1 << LedBit);
    }
//Envío de mensaje para que se inicie el procesamiento en el procesador MB1
   {
        XStatus status;

     //     print("\r\nRunning MailboxExample() for MAILBOX_0...\r\n");
        status = Mailbox(XPAR_MAILBOX_0_TESTAPP_ID, envia);
        if (status == 0)
           {   // Mensaje enviado
           }
       else
           {   //mail no enviado
           }
    }
```

Luego de finalizada la ejecución del algoritmo de procesamiento en el procesador MB0, se espera el mensaje del procesador MB1 que confirme el fin del procesamiento y almacenamiento en memoria compartida. Por último, se pone en bajo el pin utilizado para medir el tiempo de procesamiento.

```
   while(status != 0)   //espera el mensaje de finalización del procesador MB1
          status = Mailbox(XPAR_MAILBOX_0_TESTAPP_ID, recibe);
```

```
        //se pone en bajo la salida para medir el tiempo de procesamiento
         XGpio_DiscreteWrite(&GpioOutput, LED_CHANNEL, 0x0);
```

A continuación se describe el código de cada uno de los algoritmos a ejecutar en el procesador MB0. Básicamente lo que se muestra es la ejecución de 128 ventanas validas correspondientes a una fila de la imagen obtenida como resultado del procesamiento.

Código para el algoritmo de media

```
/*****************************************************************************************
 * Procesamiento de las ventanas generadas por tres filas de la imagen
 *****************************************************************************************/
   for(i=0; i<sizeimage; i++)
      {
       if(i==0)     //se verifica si es el primer píxel de la fila para hacer un promediado parcial
           {
           pixeles_output[i]=(pixeles_input1[i]+pixeles_input2[i]+pixeles_input3[i]+2*(pixeles_input
           1[i+1]+pixeles_input2[i+1]+pixeles_input3[i+1]))/9;
            }
        else if(i==sizeimage-1) //se verifica si es el ultimo píxel de la fila para hacer un promediado
                                //parcial
         {
           pixeles_output[i]=(pixeles_input1[i]+pixeles_input2[i]+pixeles_input3[i]+2*(pixeles_input
         1[i-1]+pixeles_input2[i-1]+pixeles_input3[i-1]))/9;
         }
        else if(i>0 && i<sizeimage-1)  //si no es un píxel del borde se promedia la ventana completa
         {
           pixeles_output[i]=(pixeles_input1[i-1]+pixeles_input2[i-1]+pixeles_input3[i-1]+
          pixeles_input1[i]+pixeles_input2[i]+pixeles_input3[i]+pixeles_input1[i+1]+
           pixeles_input2[i+1]+ pixeles_input3[i+1])/9;
          }
      }
//se escribe el resultado de procesamiento en memoria compartida a partir de la fila 64 de la //
//imagen
 Xil_Mem8((u8*) (bases+8192+i*128), sizeimage, pixeles_output1,escritura);
```

Código para el algoritmo de mediana

```
/*****************************************************************************************/
 // Procesamiento de las ventanas generadas por tres filas de la imagen
 /*****************************************************************************************/

for(i=0;i<sizeimage;i++)
  {
   if(i==0)
          {temp[0]=pixeles_input1[i];
           temp[1]=pixeles_input1[i+1];
           temp[2]=pixeles_input2[i];
           temp[3]=pixeles_input2[i+1];
           temp[4]=pixeles_input3[i];
           temp[5]=pixeles_input3[i+1];
           for(n=1; n<6; n++)
              {
```

```
            for(m=0; m<6-n; m++)
              {
             if(temp[m]>temp[m+1])
                 {
                                  aux  = temp[m+1];
                                   temp[m+1] = temp[m];
                                   temp[m]   = aux;
                 }
              }
           }
         pixeles_output[i]=temp[4];
         }
 else if(i==sizeimage-1)
       {
             temp[0]=pixeles_input1[i-1];
             temp[1]=pixeles_input1[i];
             temp[2]=pixeles_input2[i-1];
             temp[3]=pixeles_input2[i];
             temp[4]=pixeles_input3[i-1];
             temp[5]=pixeles_input3[i];
          for(n=1; n<6; n++)
             {
              for(m=0; m<6-n; m++)
                 {
                   if(temp[m]>temp[m+1])
                     {
                                       aux    = temp[m+1];
                                        temp[m+1] = temp[m];
                                        temp[m]   = aux;
                     }
                 }
             }
                 pixeles_output[i]=temp[4];
         }
else if(i>0 && i<sizeimage-1)
        {
             temp[0]=pixeles_input1[i-1];
             temp[1]=pixeles_input1[i];
             temp[2]=pixeles_input1[i+1];
             temp[3]=pixeles_input2[i-1];
             temp[4]=pixeles_input2[i];
             temp[5]=pixeles_input2[i+1];
             temp[6]=pixeles_input3[i-1];
             temp[7]=pixeles_input3[i];
             temp[8]=pixeles_input3[i+1];
          for(n=1; n<9; n++)
              {
               for(m=0; m<9-n; m++)
                 {
                   if(temp[m]>temp[m+1])
                     {
                                       aux    = temp[m+1];
                                        temp[m+1] = temp[m];
                                        temp[m]   = aux;
                     }
```

```
                }
            }
          pixeles_output[i]=temp[4];
        }
    }

   XStatus status;

}//fin de bucle for

//se escribe el resultado de procesamiento en memoria compartida a partir de la fila 64 de la //
//imagen de salida
 Xil_Mem8((u8*) (bases+8192+i*128), sizeimage, pixeles_output1,escritura);
```

Código para el algoritmo de sobel

```
/*****************************************************************************************/
 // Procesamiento de las ventanas generadas por tres filas de la imagen
 /*****************************************************************************************/
  for(i=0; i<sizeimage; i++)
   {
     if(i==0)
       {
         pixeles_outputx[i]=kernelx[0][2]*pixeles_input1[i+1]+kernelx[1][2]*pixeles_input2[i+1]+
        kernelx[2][2]*pixeles_input3[i+1]+kernelx[0][0]*pixeles_input1[i+1]+kernelx[1][0]*
         pixeles_input2[i+1]+ kernelx[2][0]*pixeles_input3[i+1];

          pixeles_outputy[i]=kernely[2][0]*pixeles_input3[i+1]+kernely[2][1]*pixeles_input3[i]+
         kernely[2][2]*pixeles_input3[i+1]+kernely[0][0]*pixeles_input1[i+1]+kernely[0][1]*
         pixeles_input1[i]+kernely[0][2]*pixeles_input1[i+1];
        }
    else if(i==sizeimage-1)
          {
           pixeles_outputx[i]=kernelx[0][2]*pixeles_input1[i-1]+kernelx[1][2]* pixeles_input2[i-1]+
          kernelx[2][2]*pixeles_input3[i-1]+kernelx[0][0]*pixeles_input1[i-1]+kernelx[1][0]*
          pixeles_input2[i-1]+kernelx[2][0]*pixeles_input3[i-1];

           pixeles_outputy[i]=kernely[2][0]*pixeles_input3[i-1]+kernely[2][1]*pixeles_input3[i]+
           kernely[2][2]*pixeles_input3[i+1]+kernely[0][0]*pixeles_input1[i-1]+kernely[0][1]*
           pixeles_input1[i]+kernely[0][2]*pixeles_input1[i-1];
         }
    else if(i>0 && i<sizeimage-1)
          {
         pixeles_outputx[i]=kernelx[0][2]*pixeles_input1[i+1]+kernelx[1][2]*pixeles_input2[i+1]+
        kernelx[2][2]*pixeles_input3[i+1]+kernelx[0][0]*pixeles_input1[i-1]+kernelx[1][0]*
         pixeles_input2[i-1]+kernelx[2][0]*pixeles_input3[i-1];

        pixeles_outputy[i]=kernely[2][0]*pixeles_input3[i-1]+kernely[2][1]*pixeles_input3[i]+
         kernely[2][2]*pixeles_input3[i+1]+kernely[0][0]*pixeles_input1[i-1]+kernely[0][1]*
         pixeles_input1[i]+kernely[0][2]*pixeles_input1[i+1];
         }

     pixeles_output[i]=sqrt(pow(pixeles_outputx[i],2)+pow(pixeles_outputy[i],2));
```

```
    if(pixeles_output[i]>tmax)
          pixeles_output[i]=max;
    else
         pixeles_output[i]=0;
    XStatus status;

}// fin bucle for

//se escribe el resultado de procesamiento en memoria compartida a partir de la fila 64 de la //
//imagen de salida
 Xil_Mem8((u8*) (bases+8192+i*128), sizeimage, pixeles_output1,escritura);
```

C.2. Código Microblaze 1

Las bibliotecas y funciones que se utilizan son las mismas que en el microblaze 0. Además, se inicializan los mismos periféricos y variables. En esta sección se detallan las particularidades del código ejecutado en Microblaze 1.

C.2.1. Código de mensaje de inicio del procesamiento

Para comenzar la ejecución del algoritmo, el microblaze 1 espera el mensaje de inicio proveniente del microblaze 0.

```
  {
     XStatus status;
      while(status != 0)
      status = Mailbox(XPAR_MAILBOX_0_TESTAPP_ID,recibe);
   }
```

El código de lectura de datos desde memoria compartida es exactamente igual que en el microblaze 0, solo que aquí se lee la primer mitad de la imagen desde basei hasta basei+(sizeimage/2+1).

```
Xil_Mem8((u8*)bases, 128, pixeles_output, lectura);
```

C.2.2. Código de los algoritmos de procesamiento

El código para la ejecución de los algoritmos de mediana, media y de sobel, es idéntico al ejecutado en microblaze 0. Solo se debe tener en cuenta que al llamar a la función para almacenamiento de resultado en memoria compartida, los píxeles a almacenar se ubican a partir de la dirección base ***bases*** hasta la fila 64 inclusive.

```
Xil_Mem8((u8*)bases, 128, pixeles_output, escritura);
```

Al finalizar la ejecución del algoritmo, se debe enviar un mensaje de confirmación al microblaze 0 para que comience la transmisión de la imagen procesada hacia el PC-Host.

```
{
    XStatus status;
    status = Mailbox(XPAR_MAILBOX_0_TESTAPP_ID,envia);
```

```
   if (status == 0) {}
     else {
              print("mail falló\r\n");
             }
}
```

D. Códigos de los Coprocesadores

D.1. HDL para algoritmo de erosión dilatación y mediana

Los códigos implementados en VHDL no están 100% optimizados, de cualquier manera proveen un alto grado de concurrencia al sistema. El diseño de los mismos se realizó en base a [21] y [22]. En port se definen todas las entradas y salidas del código que implementa el algoritmo de mediana y en arquitectura Behavioral las señales internas para las operaciones intermedias de ordenamiento.

--Programa principal

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

generic (vwidth: integer:=8);
port (
      Clk : in std_logic;
      RSTn : in std_logic;
      v11 : in std_logic_vector((vwidth -1) downto 0);
      v12 : in std_logic_vector((vwidth-1) downto 0);
      v13 : in std_logic_vector((vwidth -1) downto 0);
      v21 : in std_logic_vector((vwidth -1) downto 0);
      v22 : in std_logic_vector((vwidth -1) downto 0);
      v23 : in std_logic_vector((vwidth -1) downto 0);
      v31 : in std_logic_vector((vwidth-1) downto 0);
      v32 : in std_logic_vector((vwidth -1) downto 0);
      v33 : in std_logic_vector((vwidth -1) downto 0);
      DVv : in std_logic;
      DVs : out std_logic;
      s1 : out std_logic_vector(vwidth -1 downto 0);
      s2 : out std_logic_vector(vwidth-1 downto 0);
      s3 : out std_logic_vector(vwidth -1 downto 0);
      s4 : out std_logic_vector(vwidth -1 downto 0);
      s5 : out std_logic_vector(vwidth -1 downto 0);
      s6 : out std_logic_vector(vwidth -1 downto 0);
      s7 : out std_logic_vector(vwidth -1 downto 0);
      s8 : out std_logic_vector(vwidth -1 downto 0);
      s9 : out std_logic_vector(vwidth -1 downto 0)
   );
end ordenamiento_3x3;

architecture Behavioral of ordenamiento_3x3 is
-- señales de comparación
signal c11_L: std_logic_vector((vwidth -1) downto 0);
signal c11_H: std_logic_vector((vwidth-1) downto 0);
signal c12_L: std_logic_vector((vwidth -1) downto 0);
signal c12_H: std_logic_vector((vwidth -1) downto 0);
signal c13_L: std_logic_vector((vwidth -1) downto 0);
signal c13_H: std_logic_vector((vwidth -1) downto 0);
signal c14_L: std_logic_vector((vwidth -1) downto 0);
```

```
signal c14_H: std_logic_vector((vwidth -1) downto 0);
signal c21_L: std_logic_vector((vwidth -1) downto 0);
signal c21_H: std_logic_vector((vwidth -1) downto 0);
signal c22_L: std_logic_vector((vwidth -1) downto 0);
signal c22_H: std_logic_vector((vwidth -1) downto 0);
signal c23_L: std_logic_vector((vwidth -1) downto 0);
signal c23_H: std_logic_vector((vwidth -1) downto 0);
signal c24_L: std_logic_vector((vwidth -1) downto 0);
signal c24_H: std_logic_vector((vwidth-1) downto 0);
signal c31_L: std_logic_vector((vwidth -1) downto 0);
signal c31_H: std_logic_vector((vwidth -1) downto 0);
signal c32_L: std_logic_vector((vwidth -1) downto 0);
signal c32_H: std_logic_vector((vwidth -1) downto 0);
signal c33_L: std_logic_vector((vwidth-1) downto 0);
signal c33_H: std_logic_vector((vwidth -1) downto 0);
signal c34_L: std_logic_vector((vwidth -1) downto 0);
signal c34_H: std_logic_vector((vwidth -1) downto 0);
signal c41_L: std_logic_vector((vwidth -1) downto 0);
signal c41_H: std_logic_vector((vwidth -1) downto 0);
signal c42_L: std_logic_vector((vwidth -1) downto 0);
signal c42_H: std_logic_vector((vwidth -1) downto 0);
signal c43_L: std_logic_vector((vwidth -1) downto 0);
signal c43_H: std_logic_vector((vwidth -1) downto 0);
signal c4a1_L: std_logic_vector((vwidth -1) downto 0);
signal c4a1_H: std_logic_vector((vwidth -1) downto 0);
signal c4a2_L: std_logic_vector((vwidth -1) downto 0);
signal c4a2_H: std_logic_vector((vwidth -1) downto 0);
signal c4b0_L: std_logic_vector((vwidth-1) downto 0);
signal c4b0_H: std_logic_vector((vwidth -1) downto 0);
signal c4b1_L: std_logic_vector((vwidth -1) downto 0);
signal c4b1_H: std_logic_vector((vwidth -1) downto 0);
signal c4b2_L: std_logic_vector((vwidth -1) downto 0);
signal c4b2_H: std_logic_vector((vwidth -1) downto 0);
signal c51_L: std_logic_vector((vwidth -1) downto 0);
signal c51_H: std_logic_vector((vwidth -1) downto 0);
signal c61_L: std_logic_vector((vwidth -1) downto 0);
signal c61_H: std_logic_vector((vwidth -1) downto 0);
signal c71_L: std_logic_vector((vwidth -1) downto 0);
signal c71_H: std_logic_vector((vwidth -1) downto 0);
signal c81_L: std_logic_vector((vwidth -1) downto 0);
signal c81_H: std_logic_vector((vwidth -1) downto 0);
signal c91_L: std_logic_vector((vwidth-1) downto 0);
signal c91_H: std_logic_vector((vwidth -1) downto 0);
signal c101_L: std_logic_vector((vwidth -1) downto 0);
signal c101_H: std_logic_vector((vwidth -1) downto 0);
signal c111_L: std_logic_vector((vwidth -1) downto 0);
signal c111_H: std_logic_vector((vwidth -1) downto 0);
-- señales de registros para almacenar el resultado del ordenamiento parcial y total
signal r11: std_logic_vector((vwidth -1) downto 0);
signal r21: std_logic_vector((vwidth -1) downto 0);
signal r31: std_logic_vector((vwidth -1) downto 0);
signal r41: std_logic_vector((vwidth-1) downto 0);
signal r42: std_logic_vector((vwidth -1) downto 0);
signal r43: std_logic_vector((vwidth -1) downto 0);
signal r4a1: std_logic_vector((vwidth -1) downto 0);
```

```
signal r4a2: std_logic_vector((vwidth -1) downto 0);
signal r4a3: std_logic_vector((vwidth-1) downto 0);
signal r4a4: std_logic_vector((vwidth -1) downto 0);
signal r4a5: std_logic_vector((vwidth -1) downto 0);
signal r4b1: std_logic_vector((vwidth -1) downto 0);
signal r4b4: std_logic_vector((vwidth -1) downto 0);
signal r4b5: std_logic_vector((vwidth -1) downto 0);
signal r51: std_logic_vector((vwidth -1) downto 0);
signal r52: std_logic_vector((vwidth -1) downto 0);
signal r53: std_logic_vector((vwidth -1) downto 0);
signal r54: std_logic_vector((vwidth-1) downto 0);
signal r55: std_logic_vector((vwidth -1) downto 0);
signal r56: std_logic_vector((vwidth -1) downto 0);
signal r57: std_logic_vector((vwidth -1) downto 0);
signal r61: std_logic_vector((vwidth -1) downto 0);
signal r62: std_logic_vector((vwidth-1) downto 0);
signal r63: std_logic_vector((vwidth -1) downto 0);
signal r64: std_logic_vector((vwidth -1) downto 0);
signal r65: std_logic_vector((vwidth -1) downto 0);
signal r66: std_logic_vector((vwidth -1) downto 0);
signal r67: std_logic_vector((vwidth-1) downto 0);
signal r71: std_logic_vector((vwidth -1) downto 0);
signal r72: std_logic_vector((vwidth -1) downto 0);
signal r73: std_logic_vector((vwidth -1) downto 0);
signal r74: std_logic_vector((vwidth -1) downto 0);
signal r75: std_logic_vector((vwidth-1) downto 0);
signal r76: std_logic_vector((vwidth -1) downto 0);
signal r77: std_logic_vector((vwidth -1) downto 0);
signal r81: std_logic_vector((vwidth -1) downto 0);
signal r82: std_logic_vector((vwidth -1) downto 0);
signal r83: std_logic_vector((vwidth-1) downto 0);
signal r84: std_logic_vector((vwidth -1) downto 0);
signal r85: std_logic_vector((vwidth -1) downto 0);
signal r86: std_logic_vector((vwidth -1) downto 0);
signal r87: std_logic_vector((vwidth -1) downto 0);
signal r91: std_logic_vector((vwidth -1) downto 0);
signal r92: std_logic_vector((vwidth -1) downto 0);
signal r93: std_logic_vector((vwidth -1) downto 0);
signal r94: std_logic_vector((vwidth -1) downto 0);
signal r95: std_logic_vector((vwidth -1) downto 0);
signal r96: std_logic_vector((vwidth -1) downto 0);
signal r97: std_logic_vector((vwidth -1) downto 0);
signal r101: std_logic_vector((vwidth -1) downto 0);
signal r102: std_logic_vector((vwidth -1) downto 0);
signal r103: std_logic_vector((vwidth -1) downto 0);
signal r104: std_logic_vector((vwidth -1) downto 0);
signal r105: std_logic_vector((vwidth -1) downto 0);
signal r106: std_logic_vector((vwidth -1) downto 0);
signal r107: std_logic_vector((vwidth -1) downto 0);
signal r111: std_logic_vector((vwidth -1) downto 0);
signal r112: std_logic_vector((vwidth -1) downto 0);
signal r113: std_logic_vector((vwidth -1) downto 0);
signal r114: std_logic_vector((vwidth -1) downto 0);
signal r115: std_logic_vector((vwidth -1) downto 0);
signal r116: std_logic_vector((vwidth-1) downto 0);
```

```
signal r117: std_logic_vector((vwidth -1) downto 0);

-- señales DV para coordinación
signal ddddddddddddDV: std_logic:='0';
signal ddddddddddddDV: std_logic;
signal ddddddddddDV: std_logic;
signal ddddddddddDV: std_logic ;
signal dddddddddDV: std_logic;
signal ddddddddDV: std_logic;
signal dddddddDV: std_logic;
signal ddddddDV: std_logic;
signal dddddDV: std_logic;
signal ddddDV: std_logic;
signal dddDV: std_logic;
signal ddDV: std_logic;
signal dDV: std_logic;

begin
  process(Clk,RSTn)
    begin
      if RSTn = '0' then              --se inicializa todo en cero
         c11_L <= (others=>'0');
         c11_H <= (others=>'0');
         c12_L <= (others=>'0');
         c12_H <= (others=>'0');
         c13_L <= (others=>'0');
         c13_H <= (others=>'0');
         c14_L <= (others=>'0');
         c14_H <= (others=>'0');
         c21_L <= (others=>'0');
         c21_H <= (others=>'0');
         c22_L <= (others=>'0');
         c22_H <= (others=>'0');
         c23_L <= (others=>'0');
         c23_H <= (others=>'0');
         c24_L <= (others=>'0');
         c24_H <= (others=>'0');
         c31_L <= (others=>'0');
         c31_H <= (others=>'0');
         c32_L <= (others=>'0');
         c32_H <= (others=>'0');
         c33_L <= (others=>'0');
         c33_H <= (others=>'0');
         c34_L <= (others=>'0');
         c34_H <= (others=>'0');
         c41_L <= (others=>'0');
         c41_H <= (others=>'0');
         c42_L <= (others=>'0');
         c42_H <= (others=>'0');
         c43_L <= (others=>'0');
         c43_H <= (others=>'0');
         c4a1_L <= (others=>'0');
         c4a1_H <= (others=>'0');
         c4a2_L <= (others=>'0');
         c4a2_H <= (others=>'0');
```

```
c4b0_L <= (others=>'0');
c4b0_H <= (others=>'0');
c4b1_L <= (others=>'0');
c4b1_H <= (others=>'0');
c4b2_L <= (others=>'0');
c4b2_H <= (others=>'0');
c51_L <= (others=>'0');
c51_H <= (others=>'0');
c61_L <= (others=>'0');
c61_H <= (others=>'0');
c71_L <= (others=>'0');
c71_H <= (others=>'0');
c81_L <= (others=>'0');
c81_H <= (others=>'0');
c91_L <= (others=>'0');
c91_H <= (others=>'0');
c101_L <= (others=>'0');
c101_H <= (others=>'0');
c111_L <= (others=>'0');
c111_H <= (others=>'0');
r11 <= (others=>'0');
r21 <= (others=>'0');
r31 <= (others=>'0');
r41 <= (others=>'0');
r42 <= (others=>'0');
r43 <= (others=>'0');
r4a1 <= (others=>'0');
r4a2 <= (others=>'0');
r4a3 <= (others=>'0');
r4a4 <= (others=>'0');
r4a5 <= (others=>'0');
r4b1 <= (others=>'0');
r4b4 <= (others=>'0');
r4b5 <= (others=>'0');
r51 <= (others=>'0');
r52 <= (others=>'0');
r53 <= (others=>'0');
r54 <= (others=>'0');
r55 <= (others=>'0');
r56 <= (others=>'0');
r57 <= (others=>'0');
r61 <= (others=>'0');
r62 <= (others=>'0');
r63 <= (others=>'0');
r64 <= (others=>'0');
r65 <= (others=>'0');
r66 <= (others=>'0');
r67 <= (others=>'0');
r71 <= (others=>'0');
r72 <= (others=>'0');
r73 <= (others=>'0');
r74 <= (others=>'0');
r75 <= (others=>'0');
r76 <= (others=>'0');
r77 <= (others=>'0');
```

```
        r81 <= (others=>'0');
        r82 <= (others=>'0');
        r83 <= (others=>'0');
        r84 <= (others=>'0');
        r85 <= (others=>'0');
        r86 <= (others=>'0');
        r87 <= (others=>'0');
        r91 <= (others=>'0');
        r92 <= (others=>'0');
        r93 <= (others=>'0');
        r94 <= (others=>'0');
        r95 <= (others=>'0');
        r96 <= (others=>'0');
        r97 <= (others=>'0');
        r101 <= (others=>'0');
        r102 <= (others=>'0');
        r103 <= (others=>'0');
        r104 <= (others=>'0');
        r105 <= (others=>'0');
        r106 <= (others=>'0');
        r107 <= (others=>'0');
        r111 <= (others=>'0');
        r112 <= (others=>'0');
        r113 <= (others=>'0');
        r114 <= (others=>'0');
        r115 <= (others=>'0');
        r116 <= (others=>'0');
        r117 <= (others=>'0');
        s1 <= (others=>'0');
        s2 <= (others=>'0');
        s3 <= (others=>'0');
        s4 <= (others=>'0');
        s5 <= (others=>'0');
        s6 <= (others=>'0');
        s7 <= (others=>'0');
        s8 <= (others=>'0');
        s9 <= (others=>'0');
        ddddddddddddDV<= '0';
        ddddddddddddDV<= '0';
        ddddddddddDV<= '0';
        dddddddddDV<= '0';
        ddddddddDV<= '0';
        dddddddDV<= '0';
        ddddddDV<= '0';
        dddddDV<= '0';
        ddddDV<= '0';
        dddDV<= '0';
        ddDV<= '0';
        dDV<= '0';
        DVs <= '0';
    elsif rising_edge(Clk) then
        if DVv = '1' then
        -- nivel 1 de comparación
          if v11 < v12 then
                    c11_L <= v11;
```

```
            c11_H <= v12;
    else
            c11_L <= v12;
            c11_H <= v11;
    end if;

    if v13 < v21 then
            c12_L <= v13;
            c12_H <= v21;
    else
            c12_L <= v21;
            c12_H <= v13;
    end if;
    if v22 < v23 then
            c13_L <= v22;
            c13_H <= v23;
    else
            c13_L <= v23;
            c13_H <= v22;
    end if;
    if w31 < v32 then
            c14_L <= v31;
            c14_H <= v32;
    else
            c14_L <= v32;
            c14_H <= v31;
    end if;
    r11 <= v33;
-- nivel 2 de comparación
    if c11_L < c12_L then
            c21_L <= c11_L;
            c21_H <= c12_L;
    else
            c21_L <= c12_L;
            c21_H <= c11_L;
    end if;
    if c11_H < c12_H then
            c22_L <= c11_H;
            c22_H <= c12_H;
    else
            c22_L <= c12_H;
            c22_H <= c11_H;
    end if;
    if c13_L < c14_L then
            c23_L <= c13_L;
            c23_H <= c14_L;
    else
            c23_L <= c14_L;
            c23_H <= c13_L;
    end if;
    if c13_H < c14_H then
            c24_L <= c13_H;
            c24_H <= c14_H;
    else
            c24_L <= c14_H;
```

```
            c24_H <= c13_H;
  end if;
  r21 <= r11;
-- nivel 3 de comparación
  if c21_L < c23_L then
            c31_L <= c21_L;
            c31_H <= c23_L;
  else
            c31_L <= c23_L;
            c31_H <= c21_L;
  end if;
  if c21_H < c23_H then
            c32_L <= c21_H;
            c32_H <= c23_H;
  else
            c32_L <= c23_H;
            c32_H <= c21_H;
  end if;
  if c22_L < c24_L then
            c33_L <= c22_L;
            c33_H <= c24_L;
  else
            c33_L <= c24_L;
            c33_H <= c22_L;
  end if;
  if c22_H < c24_H then
            c34_L <= c22_H;
            c34_H <= c24_H;
  else
            c34_L <= c24_H;
            c34_H <= c22_H;
  end if;
  r31 <= r21;
-- nivel 4 de comparación
  r41 <= c31_L;
  if c31_H < c32_L then
            c41_L <= c31_H;
            c41_H <= c32_L;
  else
            c41_L <= c32_L;
            c41_H <= c31_H;
  end if;
  if c32_H < c33_L then
            c42_L <= c32_H;
            c42_H <= c33_L;
  else
            c42_L <= c33_L;
            c42_H <= c32_H;
  end if;
  if c33_H < c34_L then
            c43_L <= c33_H;
            c43_H <= c34_L;
  else
            c43_L <= c34_L;
            c43_H <= c33_H;
```

```
    end if;
    r42 <= c34_H;
    r43 <= r31;
  -- nivel 4a de comparación
    r4a1 <= r41;
    if c41_L < c42_H then
            c4a1_L <= c41_L;
            c4a1_H <= c42_H;
    else
            c4a1_L <= c42_H;
            c4a1_H <= c41_L;
    end if;
    if c41_H < c42_L then
            c4a2_L <= c41_H;
            c4a2_H <= c42_L;
    else
            c4a2_L <= c42_L;
            c4a2_H <= c41_H;
    end if;
    r4a2 <= c43_L;
    r4a3 <= c43_H;
    r4a4 <= r42;
    r4a5 <= r43;
-- nivel 4b de comparación
    r4b1 <= r4a1;
    if c4a1_L < c4a2_L then
            c4b0_L <= c4a1_L;
            c4b0_H <= c4a2_L;
    else
            c4b0_L <= c4a2_L;
            c4b0_H <= c4a1_L;
    end if;
    if c4a2_H < r4a2 then
            c4b1_L <= c4a2_H;
            c4b1_H <= r4a2;
    else
            c4b1_L <= r4a2;
            c4b1_H <= c4a2_H;
    end if;
    if c4a1_H < r4a3 then
            c4b2_L <= c4a1_H;
            c4b2_H <= r4a3;
    else
            c4b2_L <= r4a3;
            c4b2_H <= c4a1_H;
    end if;
    r4b4 <= r4a4;
    r4b5 <= r4a5;
  -- nivel 5 de comparación
    if r4b1 < r4b5 then
            c51_L <= r4b1;
            c51_H <= r4b5;
    else
            c51_L <= r4b5;
            c51_H <= r4b1;
```

```
  end if;
  r51 <= c4b0_L;
  r52 <= c4b0_H;
  r53 <= c4b1_L;
  r54 <= c4b1_H;
  r55 <= c4b2_L;
  r56 <= c4b2_H;
  r57 <= r4b4;
-- nivel 6 de comparación
  if r51 < c51_H then
            c61_L <= r51;
            c61_H <= c51_H;
  else
            c61_L <= c51_H;
          c61_H <= r51;
  end if;
  r61 <= c51_L; -- L
  r62 <= r52;
  r63 <= r53;
  r64 <= r54;
  r65 <= r55;
  r66 <= r56;
  r67 <= r57;
-- nivel 7 de comparación
  if r62 < c61_H then
            c71_L <= r62;
            c71_H <= c61_H;
  else
            c71_L <= c61_H;
            c71_H <= r62;
  end if;
            r71 <= r61; -- L
            r72 <= c61_L; -- 2L
            r73 <= r63;
            r74 <= r64;
            r75 <= r65;
            r76 <= r66;
            r77 <= r67;
-- nivel 8 de comparación
  if r73 < c71_H then
            c81_L <= r73;
            c81_H <= c71_H;
  else
            c81_L <= c71_H;
            c81_H <= r73;
  end if;
  r81 <= r71; -- L
  r82 <= r72; -- 2L
  r83 <= c71_L; -- 3L
  r84 <= r74;
  r85 <= r75;
  r86 <= r76;
  r87 <= r77;
-- nivel 9 de comparación
  if r84 < c81_H then
```

```
            c91_L <= r84;
            c91_H <= c81_H;
    else
            c91_L <= c81_H;
            c91_H <= r84;
    end if;
            r91 <= r81; -- L
            r92 <= r82; -- 2L
            r93 <= r83; -- 3L
            r94 <= c81_L; -- 4L
            r95 <= r85;
            r96 <= r86;
            r97 <= r87;
  -- nivel 10 de comparación
    if r95 < c91_H then
            c101_L <= r95;
            c101_H <= c91_H;
    else
            c101_L <= c91_H;
            c101_H <= r95;
    end if;
    r101 <= r91; -- L
    r102 <= r92; -- 2L
    r103 <= r93; -- 3L
    r104 <= r94; -- 4L
    r105 <= c91_L; -- M
    r106 <= r96;
    r107 <= r97;
  -- nivel 11 de comparación
    if r106 < c101_H then
            c111_L <= r106;
            c111_H <= c101_H;
    else
            c111_L <= c101_H;
            c111_H <= r106;
    end if;
    r111 <= r101; -- L
    r112 <= r102; -- 2L
    r113 <= r103; -- 3L
    r114 <= r104; -- 4L
    r115 <= r105; -- M
    r116 <= c101_L; -- 4L
    r117 <= r107;
  -- nivel 12 de comparación
    if r117 < c111_H then
            s8 <= r117; -- 2H
            s9 <= c111_H; -- H
    else
            s8 <= c111_H; -- 2H
            s9 <= r117; -- H
    end if;
 -- resultado del ordenamiento en s1….s7
    s1 <= r111; -- L
    s2 <= r112; -- 2L
    s3 <= r113; -- 3L
```

```
            s4 <= r114; -- 4L
            s5 <= r115; -- M
            s6 <= r116; -- 4H
            s7 <= c111_L; -- 3H
            ddddddddddddDV<= ddddddddddddDV;
            ddddddddddddDV<= ddddddddddddDV;
            ddddddddddDV<= ddddddddddDV;
            ddddddddddDV<= ddddddddddDV;
            dddddddddDV<= dddddddddDV;
            ddddddddDV<= ddddddddDV;
            dddddddDV<= dddddddDV;
            ddddddDV<= ddddddDV;
            dddddDV<= dddddDV;
            ddddDV<= ddddDV;
            dddDV<= dddDV;
            ddDV<= ddDV;
            dDV<= ddDV;
            DVs <= dDV;
        end if;
        if DVv = '1' then
            ddddddddddddddDV<= '1';
        end if;
    end if;
  end process;
end Behavioral;
```

D.2. HDL para algoritmo de media

```
use IEEE.numeric_std.all;

    Entity media is
        port (
        sample: in integer range 0 to 255;
        clk: in STD_LOGIC;
        fout: out INTEGER range 0 to 255
         );
    end media;

    Architecture media_arch of media is
    Begin
       Process ( clk )
          Variable p0,p1,p2,p3 : integer range 0 to 255;
          Variable p4,p5,p6,p7,p8 : integer range 0 to 255;
        Begin
            Wait until ( clk'event and clk' = '1' );
          p8:=p7;
          p7:=p6;
          p6:=p5;
          p4:=p3;
          p3:=p2;
           p2:=p1;
           p1:=p0;
           p0:=sample; --en cada flanco ascendente de clock ingresa un nuevo dato
```

```
        --cuando ingresan los 9 datos de las ventana a procesar se realiza el promedio

          fout<= (p0+p1+p2+p3+p4+p5+p6+p7+p8)/9;

       end process;
  end media_arch;
```

D.3. HDL de convolución para algoritmo de sobel

-- Package

```
package conv_3x3_pkg is
-- Las constantes kx definen el kernel a ser usado en la operación de convolución
-- El valor kx está en el rando de -128<kx<128
constant k0 : std_logic_vector(7 downto 0):=std_logic_vector(to_signed(-1,8));
constant k1 : std_logic_vector(7 downto 0):=std_logic_vector(to_signed(-2,8));
constant k2 : std_logic_vector(7 downto 0):=std_logic_vector(to_signed(-1,8));
constant k3 : std_logic_vector(7 downto 0):=std_logic_vector(to_signed(0,8));
constant k4 : std_logic_vector(7 downto 0):=std_logic_vector(to_signed(0,8));
constant k5 : std_logic_vector(7 downto 0):=std_logic_vector(to_signed(0,8));
constant k6 : std_logic_vector(7 downto 0):=std_logic_vector(to_signed(1,8));
constant k7 : std_logic_vector(7 downto 0):=std_logic_vector(to_signed(2,8));
constant k8 : std_logic_vector(7 downto 0):=std_logic_vector(to_signed(1,8));
constant vwidth : integer := 8;
constant order : integer := 1;
constant num_cols : integer := 128;
constant num_rows : integer := 128;
end conv_3x3_pkg;
```

--Programa principal

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.conv_3x3_pkg.all;   --se incluye el package donde se define el kernel de sobel

entity convolucion3x3 is
    port (
          Clk : in std_logic;
          RSTn : in std_logic;
          D : in std_logic_vector(vwidth-1 downto 0);
          Dout : out std_logic_vector((vwidth*2)+1 downto 0);
          DV : out std_logic
           );
end convolucion3x3;

architecture Behavioral of convolucion3x3 is
--señales que representan los píxeles de la ventana a procesar
signal v11: std_logic_vector((vwidth -1) downto 0);
signal v12: std_logic_vector((vwidth -1) downto 0);
signal v13: std_logic_vector((vwidth -1) downto 0);
signal v21: std_logic_vector((vwidth -1) downto 0);
signal v22: std_logic_vector((vwidth -1) downto 0);
signal v23: std_logic_vector((vwidth -1) downto 0);
```

```
signal v31: std_logic_vector((vwidth -1) downto 0);
signal v32: std_logic_vector((vwidth -1) downto 0);
signal v33: std_logic_vector((vwidth -1) downto 0);
signal DVv: std_logic;

component ventana_3x3
generic (
          vwidth: integer:=8
         );

port (
         Clk : in std_logic;
         RSTn : in std_logic;
         D : in std_logic_vector(vwidth-1 downto 0);
         v11 : out std_logic_vector(vwidth -1 downto 0);
         v12 : out std_logic_vector(vwidth -1 downto 0);
         v13 : out std_logic_vector(vwidth -1 downto 0);
         v21 : out std_logic_vector(vwidth-1 downto 0);
         v22 : out std_logic_vector(vwidth -1 downto 0);
         v23 : out std_logic_vector(vwidth -1 downto 0);
         v31 : out std_logic_vector(vwidth -1 downto 0);
         v32 : out std_logic_vector(vwidth -1 downto 0);
         v33 : out std_logic_vector(vwidth-1 downto 0);
         DV : out std_logic:='0'
        );
end component ventana_3x3;

-- 16 bits para operaciones de 8bit x 8bit  más 1 bit para signo
signal m0: signed((vwidth*2) downto 0):=(others=>'0');
signal m1: signed((vwidth*2) downto 0):=(others=>'0');
signal m2: signed((vwidth*2) downto 0):=(others=>'0');
signal m3: signed((vwidth*2) downto 0):=(others=>'0');
signal m4: signed((vwidth*2) downto 0):=(others=>'0');
signal m5: signed((vwidth*2) downto 0):=(others=>'0');
signal m6: signed((vwidth*2) downto 0):=(others=>'0') ;
signal m7: signed((vwidth*2) downto 0):=(others=>'0');
signal m8: signed((vwidth*2) downto 0):=(others=>'0');
signal a10: signed((vwidth*2)+1 downto 0):=(others=>'0');
signal a11: signed((vwidth*2)+1 downto 0):=(others=>'0');
signal a12: signed((vwidth*2)+1 downto 0):=(others=>'0');
signal a13: signed((vwidth*2)+1 downto 0):=(others=>'0');
signal a14: signed((vwidth*2)+1 downto 0):=(others=>'0');
signal a20: signed((vwidth*2)+2 downto 0):=(others=>'0');
signal a21: signed((vwidth*2)+2 downto 0):= (others=>'0');
signal a22: signed((vwidth*2)+2 downto 0):=(others=>'0');
signal a30: signed((vwidth*2)+3 downto 0):=(others=>'0');
signal a31: signed((vwidth*2)+3 downto 0):=(others=>'0');
signal a40: signed((vwidth*2)+4 downto 0):=(others=>'0');
signal d0: signed((vwidth*2)+1 downto 0):=(others=>'0');

component contador_rc
generic (
         num_cols: integer:=128;
         num_rows: integer:=128
          );
```

```
 port (
          Clk : in std_logic;
          RSTn : in std_logic;
          En : in std_logic;
          ColPos : out integer;
          RowPos : out integer
        );
end component contador_rc;

signal ColPos: integer:=0;
signal RowPos: integer:=0;
signal ColPos_c: integer:=0;  -- corrector de posición
signal RowPos_c: integer:=0;
signal rt1: integer:=0;
signal rt2: integer:=0;
signal rt3: integer:=0;
signal rt4: integer:=0;
signal rt5: integer:=0;
signal rt6: integer:=0;
signal rt7: integer:=0;
signal rt8: integer:=0;
signal flag: std_logic:='0';

begin

window_3x3x: ventana_3x3
generic map (
                    vwidth => 8
                )
port map (
          Clk => Clk,
          RSTn => RSTn,
          D => D,
          v11 => v11,
          v12 => v12,
          v13 => v13,
          v21 => v21,
          v22 => v22,
          v23 => v23,
          v31 => v31,
          v32 => v32,
          v33 => v33,
          DV => DVv
          );

rc_counterx: contador_rc
generic map (
                num_cols => 128,
                num_rows => 128
                )
port map (
            Clk => Clk,
            RSTn => RSTn,
            En => RSTn,
            ColPos => ColPos,
```

```
            RowPos => RowPos
          );

convproc: process(Clk,RSTn)
begin
   if RSTn = '0' then
        m0 <= (others=>'0');
        m1 <= (others=>'0');
        m2 <= (others=>'0');
        m3 <= (others=>'0');
        m4 <= (others=>'0');
        m5 <= (others=>'0');
        m6 <= (others=>'0');
        m7 <= (others=>'0');
        m8 <= (others=>'0');
        a10 <= (others=>'0');
        a11 <= (others=>'0');
        a12 <= (others=>'0');
        a13 <= (others=>'0');
        a14 <= (others=>'0');
        a20 <= (others=>'0');
        a21 <= (others=>'0');
        a22 <= (others=>'0');
        a30 <= (others=>'0');
        a31 <= (others=>'0');
        a40 <= (others=>'0');
        d0 <= (others=>'0');
        Dout <= (others=>'0');
        DV <= '0';
        ColPos_c <= 0;
        rt1 <= 0;
        rt2 <= 0;
        rt3 <= 0;
        rt4 <= 0;
        rt5 <= 0;
        rt6 <= 0;
        rt7 <= 0;
        rt8 <= 0;
        RowPos_c <= 0;
        flag<= '0';
   elsif rising_edge(Clk) then
   -- corrección de contador
        ColPos_c <= ((ColPos-8) mod 128);
        rt1 <= ((RowPos-1) mod 128);
        rt2 <= rt1;
        rt3 <= rt2;
        rt4 <= rt3;
        rt5 <= rt4;
        rt6 <= rt5;
        rt7 <= rt6;
        rt8 <= rt7;
        RowPos_c <= rt8;
   -- detección de borde de imagen
        if (ColPos_c = num_cols-1) or (RowPos_c = num_rows-1) or (ColPos_c= num_cols-2) or
        (RowPos_c = 0) then
```

```
                Dout <= (others=>'0');
            end if;
            if DVv = '1' then
          --Multiplicaciones  entre  píxeles de la ventana* píxeles del kernel
          -- Esto podría optimizarse mediante el uso de multiplicadores de hardware
                m0 <= signed('0'&v11)*signed(k0);
                m1 <= signed('0'&v12)*signed(k1);
                m2 <= signed('0'&v13)*signed(k2);
                m3 <= signed('0'&v21)*signed(k3);
                m4 <= signed('0'&v22)*signed(k4);
                m5 <= signed('0'&v23)*signed(k5);
                m6 <= signed('0'&v31)*signed(k6);
                m7 <= signed('0'&v32)*signed(k7);
                m8 <= signed('0'&v33)*signed(k8);
            -- se van sumando los resultados de las multiplicaciones
                a10 <= (m0(16)&m0)+m1;
                a11 <= (m2(16)&m2)+m3;
                a12 <= (m4(16)&m4)+m5;
                a13 <= (m6(16)&m6)+m7;
                a14 <= m8(16)&m8;
                a20 <= (a10(17)&a10)+a11;
                a21 <= (a12(17)&a12)+a13;
                a22 <= a14(17)&a14;
                a30 <= (a20(18)&a20)+a21;
                a31 <= a22(18)&a22;
            --el resultado de las sumas se guarda en a40
                a40 <= (a30(19)&a30)+a31;
                -- en d0 se guarda el resultado final
                -- que se obtiene de dividir por 8 mediante desplazamientos
                d0 <= a40(20 downto 3);
                if (ColPos_c = num_cols-1) or (RowPos_c = num_rows-1) or (ColPos_c =
                num_cols-2) or (RowPos_c = 0) then
                    Dout <= (others=>'0');
                else
                    Dout <= std_logic_vector(d0);
                end if;
            end if;
            if ColPos >= 8 and RowPos >= 1 then
                DV <= '1';
                flag<= '1';
            elsif flag = '1' then
                DV <= '1';
            else
                DV <= '0';
            end if;
        end if; --end if ESTn='0'
    end process;
end Behavioral;
```