Operador morfológico pecstrum en FPGA para aplicaciones de biometría

por

Miguel Angel Moreno Cedeño

Tesis sometida como requisito parcial
para obtener el grado de

MAESTRO EN CIENCIAS EN LA ESPECIALIDAD DE ELECTRÓNICA

en el

Instituto Nacional de Astrofísica, Óptica y Electrónica
Febrero 2011
Tonantzintla, Puebla

Supervisada por:

Dr. Juan Manuel Ramírez Cortés
Investigador Titular del INAOE

Abstract.

Biometric recognition has gained importance in environments that require people identification. Biometric system aim for people identification using different unique, physiological, time- invariant characteristics for each person such as digital prints, face, iris and retina detection, hand geometry, voice, hand shape, palm print. The hand shape based techniques use biometric characteristic such as finger and palm length and width, fingers to palm size ratio, palm print information, palm shape, joint finger width or a combination between them. One of these techniques uses the pattern spectrum as a characteristic extractor to obtain hand shape quantitative information. Position and rotation invariant property of pecstrum allows the user to place the hand without extra restrictions. The Pattern spectrum consists on the successive application of binary erosion and dilatation based opening filters, using in each step a growing structuring element. The erosion and dilatation functions used in pecstrum for large structuring elements are computationally intensive, the algorithms use long time to execute, and it causes an excessive time computing. The use of an FPGA allows to obtain a compact system since the complete electronic for control and image processing may be placed in one circuit. This thesis raises the implementation of the pecstrum algorithm on a FPGA to process binary images with the purpose of offering a competitive calculation time.

Resumen.

En la actualidad los sistemas basados en reconocimiento biométrico han cobrado gran relevancia en entornos que requieren la identificación de usuarios. Los sistemas biométricos tienen como objetivo la identificación utilizando diferentes características fisiológicas invariantes en el tiempo y únicas en cada individuo, tales como huellas digitales, rostro, iris, retina, geometría de la mano, voz, forma de la mano, huellas de la palma. Las técnicas basadas en la forma de la mano usan características biométricas tales como longitud y anchura de los dedos y las palmas, relación de aspecto de la palma a los dedos, información de la impresión de la palma, contorno de la palma, anchura de los dedos en las articulaciones o una combinación de ellas. Una de estas técnicas utiliza el espectro de patrones (pecstrum) como un extractor de características para obtener información cuantitativa de la forma de la mano. La propiedad de invarianza a la rotación y a la posición del pecstrum, permite al usuario colocar naturalmente la mano sin restricciones adicionales. El Espectro de Patrones consiste en la aplicación sucesiva de filtros de apertura basados en operaciones de dilatación y erosión binaria, utilizando en cada paso un elemento estructurante con dimensión creciente. Las funciones de erosión y dilatación utilizadas en el pecstrum para valores grandes del elemento estructurante resultan computacionalmente intensas, los algoritmos utilizan mucho tiempo para ejecutarse lo que produce un tiempo excesivo de cómputo. El uso del FPGA permite obtener un sistema compacto ya que toda la parte electrónica de control y procesamiento de imagen puede quedar alojada en un solo circuito. Esta tesis plantea la implementación del pecstrum en un FPGA para procesar imágenes binarias cuyo propósito es ofrecer un tiempo de cálculo aceptable.

Agradecimientos:

Mi más sincero agradecimiento a mi director de tesis, el Dr. Juan Manuel Ramírez Cortés, por su apoyo y comprensión.

Gracias a cada uno de los maestros que participaron en mi desarrollo profesional en mi estancia en la maestría. Así también al Instituto Nacional de Astrofísica, Óptica y Electrónica, por las facilidades prestadas, biblioteca, cubículo, fotocopias, etc.

A mis compañeros de instrumentación, por brindarme su ayuda y amistad.

En general quisiera agradecer a todas y cada una de las personas que han vivido conmigo la realización de esta tesis, desde los más profundo de mi corazón les agradezco el haberme brindado todo el apoyo, colaboración, ánimo y sobre todo cariño y amistad.

Dedico esta tesis:

A mi hijo que es mi motivo principal para vivir y seguir siempre adelante.

A mi esposa.

A mis padres y hermanos

Índice

Capitulo 1.

Introduccion.

Introducción.

En esta tesis se plantea la implementación en hardware del algoritmo pecstrum, para procesamiento en tiempo real y orientado a aplicaciones de biometría, aunque se pretende que el trabajo final tenga la versatilidad para ser utilizado en otras aplicaciones.

El concepto de biometría se refiere a un rasgo biológico único e irrepetible que se puede medir, para reconocer o identificar la identidad de un individuo tales como huellas digitales, rostro, patrón de escritura, iris, retina, geometría de la mano, voz, forma de la mano, huellas de la palma, o características dinámicas como verificación en-línea de la firma. La principal ventaja de esta tecnología en comparación con los métodos clásicos utilizados comúnmente como llaves o claves, los rasgos biométricos, en general, no pueden ser prestados, robados o copiados. Muchas características fisiológicas son invariantes con el tiempo y únicas en cada individuo.

La mayoría de las investigaciones actuales en biometría se encuentran enfocadas en huellas dactilares y rostro. La fiabilidad en identificación personal utilizando el rostro es actualmente baja, así como los sistemas comerciales disponibles continúan luchando con problemas de pose, luz y expresión. La identificación utilizando huellas dactilares tiene buena aceptación, sin embargo, una gran cantidad de usuarios tales como, personas mayores y obreros no entregan huellas dactilares de buena calidad, la superficie de las huellas es pequeña y cualquier corte o cicatriz genera falsas minucias.

La identificación personal basada en el patrón de iris se ha convertido en una de las técnicas más confiables, gracias a sus características únicas, estables y accesibles [8]. La imagen del iris debe tener una apropiada cantidad de pixeles, estar bien enfocadas para que se distingan los detalles

del patrón de iris y tener un buen contraste lo cual requiere un nivel conveniente de iluminación, no demasiado alto para no molestar al usuario. Sin embargo son sistemas sofisticados e invasivos

Sistemas de verificación/identificación basados en la forma de la mano proveen un esquema alternativo. Las técnicas basadas en la forma de la mano usan características biométricas tales como longitud de los dedos y las palmas, anchura de los dedos, relación de aspecto de la palma a los dedos, información de la impresión de la palma, contorno de la palma, anchura de los dedos en las articulaciones o una combinación de ellas. En [1] se registra la forma de la mano sin importar la posición de los dedos ya que los sujetos de prueba no están restringidos a una pose o postura durante la adquisición, se requiere de una normalización de la imagen obtenida, por lo cual las imágenes capturadas en posturas y posiciones arbitrarias se llevan a una postura y pose estándar. En [4] se trabaja en sistemas de verificación basados en la impresión de la palma e integrando características de la geometría de la mano. En [5] se propone el uso del operador morfológico pecstrum (espectro de patrones) como un extractor de características para un sistema de reconocimiento basado en la forma de la mano. En [2] se utiliza la teoría de procesos de markov para modelar la generación de los patrones espectrales.

Se utiliza el espectro de patrones (pecstrum) como un extractor de características para obtener información cuantitativa de la forma de la mano. La propiedad de invarianza a la rotación y a la posición del pecstrum, permite al usuario colocar naturalmente la mano sin restricciones adicionales. Esta es una ventaja sobre otros sistemas donde se requiere una posición de mano fija o una posterior normalización de la imagen obtenida a una postura estándar.

El pecstrum consiste en la aplicación sucesiva de filtros de apertura basados en operaciones de dilatación y erosión binaria, utilizando en cada paso un elemento estructurante con dimensión creciente. Las funciones de erosión y dilatación para valores grandes del elemento estructurante resultan computacionalmente intensas lo que produce un tiempo excesivo de cómputo. Los tiempos de ejecución se pueden reducir si el pecstrum es implementado en hardware, para ello el sistema hace uso de un dispositivo lógico programable, específicamente un FPGA (Field Programmable Gate Array). Debido a la funcionalidad y flexibilidad de estos dispositivos, cada vez es más factible desarrollar sistemas para aplicaciones de procesamiento de imagen, cuyos resultados pueden ser observados en tiempo real. El uso del FPGA nos permite obtener un sistema compacto ya que toda la parte electrónica de control y procesamiento de imagen puede quedar alojada en un solo circuito.

Los FPGAs son dispositivos que permiten diseñar sistemas digitales para aplicaciones específicas. Entre las ventajas de estos dispositivos tenemos la posibilidad de poder programarlos una y otra vez, y la versatilidad de implementar prácticamente cualquier circuito en ellos. De igual forma que en los microcontroladores se carga software, en los FPGA's se carga la configuración que determina en qué circuito se va a convertir. El usuario decide en que se convertirá el dispositivo mediante su configuración.

La programación del FPGA fue realizada utilizando el lenguaje de programación VHDL, Very high speed integrated circuit (VHSIC) Hardware Description Language. VHDL está diseñado para cubrir una serie de necesidades en el proceso de diseño. En primer lugar, permite la descripción de la estructura de un diseño, que es la forma en que se descompone en sub-diseños, y cómo esos sub-diseños están interconectados. En segundo lugar, permite la especificación de la función de los diseños mediante las conocidas formas de lenguaje de programación. En tercer lugar, permite un

diseño para ser simulado antes de ser fabricado, por lo que los diseñadores pueden comparar rápidamente las alternativas y la prueba de la corrección, sin la demora y los gastos de creación de prototipos de hardware.

Esta tesis plantea la implementación del pectrum en un FPGA para procesar imágenes binarias cuyo propósito es ofrecer un tiempo de cálculo aceptable. Determinar el tiempo de cálculo para procesar el operador pecstrum sobre una imagen, usando la unidad de procesado desarrollada, así como el espacio de FPGA necesario.

La tesis esta organizada como se muestra a continuación:

En el capitulo 2, se da una descripción de la tarjeta nexys2, las características principales y los elementos que la integran. Posteriormente se describe el funcionamiento de los FPGA concentrándose principalmente en el spartan 3E de xilinx que es el que viene en la tarjeta nexys2. Más adelante se da una breve introducción al lenguaje VHDL que se utiliza para programar los FPGA. Aquí se da una breve explicación del lenguaje de programación VHDL y su uso para síntesis de FPGA’s.

El capitulo 3 contiene un pequeño repaso de lógica de conjuntos para dar paso a las operaciones morfológicas de erosión, dilatación, apertura y cerradura que son la base para el procesamiento morfológico, finalmente se define al operador pecstrum.

En el capitulo 4 se presenta el programa, los bloques que lo forman, la descripción de puertos de cada bloque y una explicación de su funcionamiento. El programa genera los valores elemento a elemento del kernel y de la submatriz de la imagen para procesarlos uno por uno, de esta manera para un kernel de 3x3 se tiene nueve operaciones para procesar un píxel, una por cada elemento del kernel, a medida que el kernel va incrementando su tamaño, el numero de operaciones también aumenta. Sin

embargo ocupa poco espacio en el FPGA y permite trabajar con kernels de dimensiones grandes. La ventaja de que ocupe pocos recursos del FPGA es que se pueden implementar varios módulos de procesamiento y procesar la misma imagen por partes y en paralelo, lo que reduce de manera considerable el tiempo de procesamiento.

Se describen las rutinas necesarias para implementar los filtros de apertura en el FPGA. Los diagramas generales, una descripción del funcionamiento de ambos circuitos y su simulación.

Finalmente se describe la interfaz de usuario que se hizo con matlab para enviar la imagen original y recibir la imagen procesada por el FPGA.

En el capitulo 5 se encuentran los resultados obtenidos y la seccion de trabajo a futuro. Se comparan los resultados de matlab con los del FPGA utilizando la intefaz de usuario, y se muestra un resumen de los recursos utilizados por el FPGA. En la parte de trabajo a futuro se presentan propuestas de configuraciones para mejorar la velocidad de procesamiento basadas bloque de procesamiento que se presenta en esta tesis.

En el apéndice vienen los listados de los bloques y sus simulaciones.

Capitulo 2.

Marco teorico.

2.01 Datos generales de la tarjeta nexys-2.

La tarjeta nexys-2 es compatible con todas las versiones de Xilinx ISE Tools incluyendo el WebPack gratis. Entre sus principales características se encuentran:

- FPGA Spartan 3E de Xilinx de 500,000 compuertas.
- USB2 para transferencia de alta velocidad (usando el software gratis Adept Suite)
- Alimentación vía USB (también se puede usar baterías).
- Puertos PS/2, VGA, serial.
- Memoria Micron PSDRAM de 16MB y memoria ROM intel StrataFlash de 16MB
- Oscilador de 50MHz y un zócalo adicional para un segundo oscilador.
- 60 I/O direccionadas a conectores de expansión.
- Plataforma Flash para configuraciones FPGA no volátiles
- 8 leds, display de 7 segmentos de 4 dígitos, 8 interruptores deslizables.

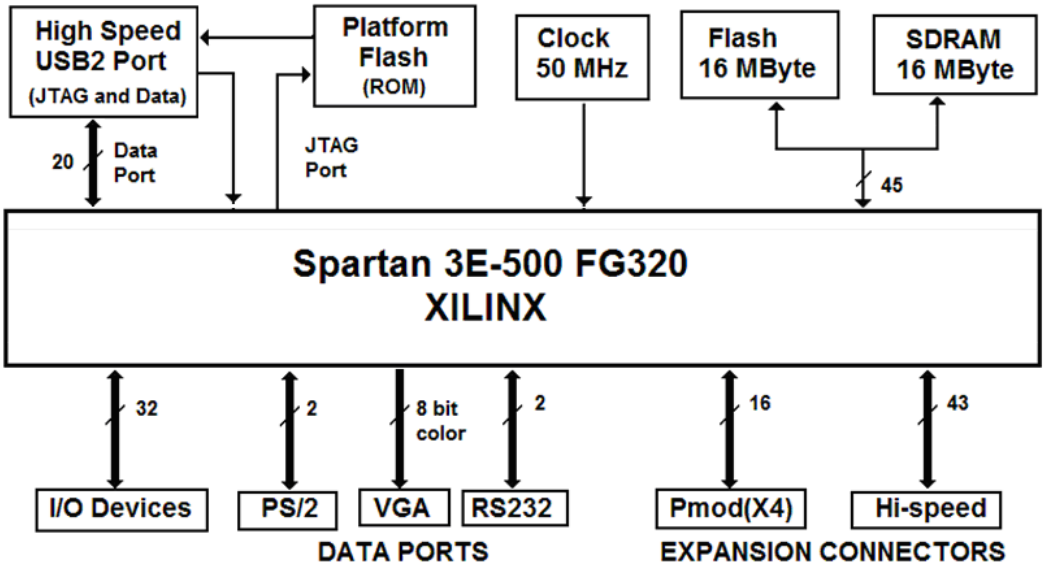

Figura 2.1. Tarjeta nexys-2.

2.02 Arquitectura del FPGA Spartan IIIE de Xilinx.

Los FPGA Spartan IIIE de Xilinx están conformadas por un conjunto de Bloques Lógicos Configurables (Configurable Logic Blocks: CLBs) rodeados por un perímetro de Bloques Programables de entrada/salida (Programmable Input/Output Blocks: IOBs).

Adicionalmente el dispositivo cuenta con secciones de memoria RAM de 360Kbits dispuestos en bloque de 18kbits y hasta 73Kbits de memoria RAM distribuida; y con 20 multiplicadores dedicados de 18 X 18 bits.

La arquitectura de la familia Spartan-3E consta de cinco principales elementos funcionales programables:

- Bloques Lógicos configurables (Configurable Logic Blocks – CLBs): Contienen flexibles Look-Up Tables (LUT) que implementan funciones lógicas y elementos de almacenamiento como flip-flops o latches. CLBs realizan una amplia variedad de funciones lógicas, así como almacenar datos.
- Bloques de entrada/salida (Input/Output Blocks – IOBs): Controlan el flujo de datos entre los pines de entrada/salida y la lógica interna del dispositivo. Cada IOB soporta flujo bidireccional más operación tri-estado y una variedad de estándares de señales, entre ellas cuatro niveles diferenciados de alto rendimiento. Registros Double Data-Rate (DDR) están incluidos.
- Bloques de memoria RAM (Block RAM): Proveen almacenamiento de datos en bloques de 18Kbits con dos puertos independientes cada uno.
- Bloques de multiplicación que aceptan dos números binarios de 18 bit como entrada y entregan uno de 36 bits.
- Administradores digitales de reloj (Digital Clock Managers – DCMs): proveen auto calibración, soluciones completamente digitales para

distribuir, retrazar, multiplicar, dividir y desfasamiento de fase de las señales de reloj.

Los elementos descritos están organizados como se muestra en la Figura 2.2. Un anillo de IOBs rodea un arreglo regular de CLBs. Cada dispositivo tiene dos columnas de memoria de RAM, compuesta por varios bloques de 18Kbit, cada uno de los cuales está asociado con un multiplicador dedicado. Los DCMs están colocados en el centro con dos en la parte superior y dos en la parte inferior del dispositivo.

Los FPGAs Spartan-3E son programados cargando los datos de configuración en latches estáticos CMOS de configuración (CCLs) reprogramables y robustos que en conjunto controlan todos los elementos funcionales y los recursos de enrutamiento. Los datos de configuración del FPGA se almacenan externamente en una PROM o algún otro medio no volátil, ya sea dentro o fuera de la tarjeta.

2.02.1 Características de las entradas / salidas.

La interfaz SelectIO del FPGA Spartan-3E es compatible con muchos estándares populares de una sola terminal y diferencial.

Soporta las siguientes normas de una sola terminal:

• 3,3 baja tensión TTL (LVTTL).

• Bajo voltaje CMOS (LVCMOS) a 3,3 V, 2,5 V, 1,8 V, 1,5 V o 1,2 V.

• 3V PCI a 33 MHz, y en algunos dispositivos 66 MHz.

• HSTL I y III a 1,8 V, de uso común en aplicaciones de memoria.

• SSTL I a 1.8V y 2.5V, usado comúnmente en aplicaciones de memoria.

Soporta los siguientes estándares diferenciales:

• LVDS.

• Bus LVDS.

• Mini-LVDS.

• RSDS.

• HSTL diferencial (1,8 V, tipo I y III).

• SSTL diferencial (I 2,5 V y 1,8 V, tipo).

• 2.5V LVPECL entradas.

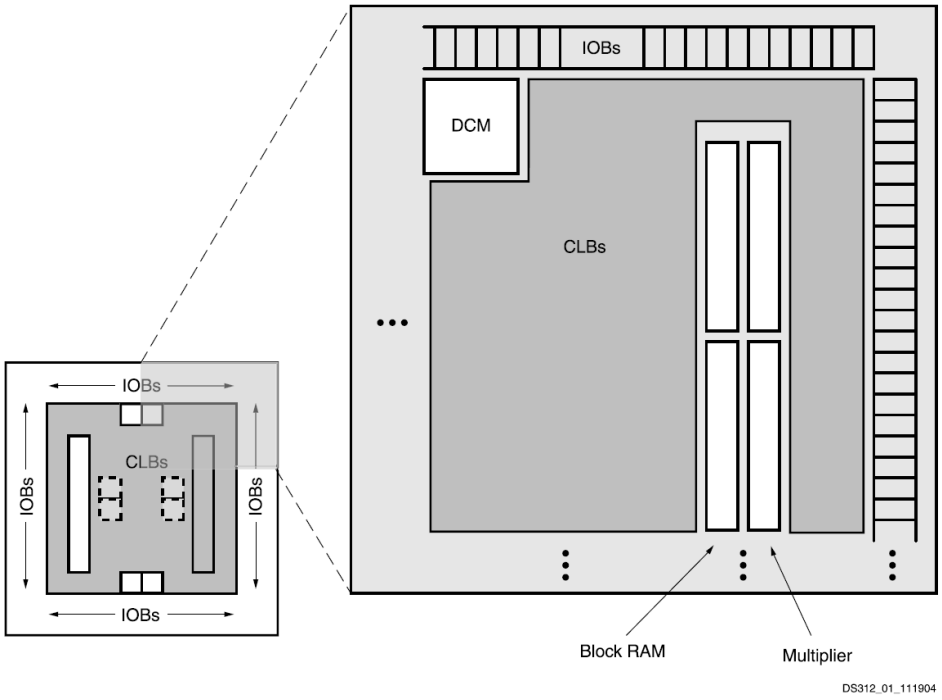

Figura 2.2: Arquitectura de la Spartan III. Imagen tomada de [14], p. 4.

A continuación se hace una descripción más detallada de cada uno de los elementos funcionales del FPGA.

2.02.2 Bloques de entrada/salida IOB (Input/Output Block).

Los bloques de entrada/salida (IOB) suministran una interfaz unidireccional o bidireccional programable entre un pin de entrada/salida y la lógica interna del FPGA. El bloque de entrada unidireccional sólo tiene un subconjunto de las capacidades completas de IOB. Por lo tanto no hay ninguna conexión o la lógica para una ruta de salida.

Un diagrama simplificado de la estructura interna de un IOB aparece en la Figura 2.3. Hay tres rutas para señales: la ruta de salida, la ruta de entrada y la ruta tri-estado. Cada ruta tiene su propio par de elementos de almacenamiento que pueden actuar tanto como registros o como latches. Las tres rutas principales son como sigue:

- La ruta de entrada lleva datos desde el pad, que está unido al pin del circuito integrado, a través de un elemento de retardo opcional programable, directamente a la línea I. Después del elemento de retardo hay rutas alternativas a través de un par de elementos de almacenamiento hacia las líneas IQ1 e IQ2. Las tres salidas del IOB conducen a la lógica interna del FPGA.

- La ruta de salida, que parte con las líneas O1 y O2, lleva datos desde la lógica interna del FPGA, a través de un multiplexor y de un driver tri-estado hacia el pad del IOB. Además de esta ruta directa, el multiplexor da la opción de insertar un par de elementos de almacenamiento.

- La ruta tri-estado determina cuando el driver de salida está en alta impedancia. Las líneas T1 y T2 llevan datos desde la lógica interna del FPGA a través de un multiplexor hacia el driver de salida. Además de esta ruta directa, el multiplexor ofrece la opción de insertar un par de elementos de almacenamiento.

Todas las rutas de señales que entran al IOB, incluidas las relacionados con los elementos de almacenamiento tienen una opción de inversión. Cualquier inversor colocado en estas rutas (en programación) es automáticamente absorbido en el IOB.

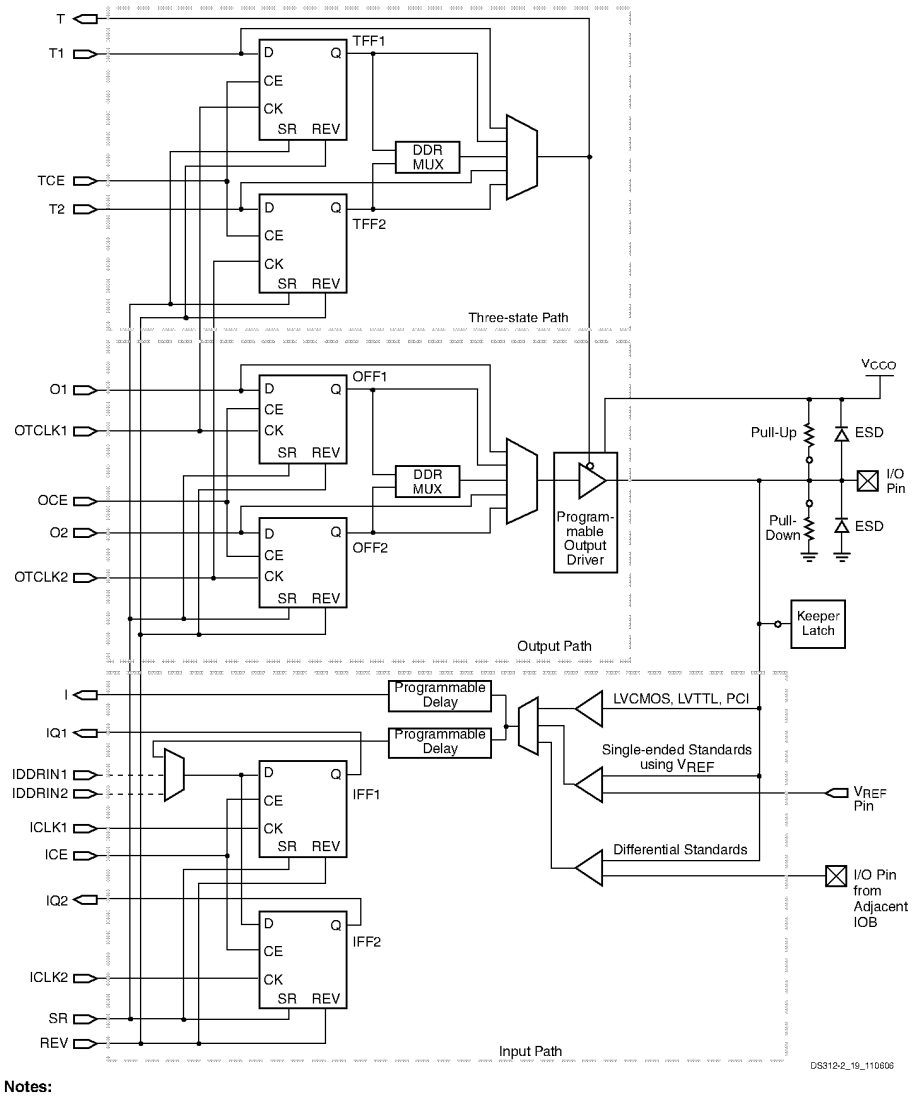

Figura 2.3: Diagrama simplificado de un IOB de la Spartan III. Imagen tomada de [14], p. 11.

Cada IOB tiene un bloque de retardo programable que, opcionalmente, retrasa la señal de entrada. Los valores de retardo se establecen solo una vez en la configuración y no son modificables durante la operación del dispositivo. El uso principal del elemento de retardo de entrada

es ajustar la trayectoria de retardo de entrada para asegurarse de que no hay requisitos de tiempo de espera cuando se utiliza la entrada del flip-flop (s) con un reloj global.

Hay tres pares de elementos de almacenamiento en cada IOB, un par por cada ruta. Es posible configurar cada uno de estos elementos como un flip-flop tipo D o como latch sensible al nivel disparado por flanco.

El par de elementos de almacenamiento tanto de la ruta de salida o del driver tri-estado pueden ser usados en conjunto con un multiplexor especial para producir transmisión de doble tasa de datos (DDR). Esto se logra tomando datos sincronizados con el flanco de subida del reloj y convirtiéndolos en bits sincronizados tanto en el flanco de subida como en el de bajada. A esta combinación de dos registros y un multiplexor se le llama flip flop tipo D de doble tasa de datos (ODDR2).

Terminación diferencial on-Chip. Los dispositivos Spartan-3E proporcionan una terminación diferencial de 120Ω on-chip entre los terminales de entrada del receptor diferencial. La entrada de terminación diferencial en el chip en dispositivos Spartan-3E potencialmente elimina la resistencia externa de terminación de 100Ω que se encuentra comúnmente en los circuitos del receptor diferencial.

Resistencias Pull-Up y Pull-Down dentro de cada IOB tienen el objetivo de forzar niveles altos o bajos en las salidas de los IOBs que no están en uso. La resistencia de Pull-Up conecta un IOB a VCCO, de manera similar la resistencia de Pull-Down conecta un IOB a tierra.

Circuito Keeper (de retención), cada I/O tiene un circuito Keeper opcional para cuidar que las líneas de un bus no floten, el circuito conserva el último nivel lógico en una línea después de que todos los drivers se han

apagado. Las resistencias Pull-up y pull-down reemplazan la configuración Keeper.

Cada IOB tiene un control de slew-rate que configura el borde de la conmutación para salidas LVCMOS y LVTTL. El atributo SLEW controla la velocidad de respuesta y se puede establecer en lento (por defecto) o rápido.

Cada salida LVCMOS y LVTTL, soporta hasta seis niveles deferentes de corrientes. Para ajustar la corriente máxima de cada salida, el atributo DRIVE se ajusta a la corriente deseada: 2, 4, 6, 8, 12 y 16. A menos que se especifique lo contrario en la aplicación del FPGA, el valor por defecto de software IOSTANDARD es LVCMOS25, slew-rate lento, y salida de 12 mA

Alta corriente de salida y un slew-rate rápido, por lo general, resultan en I/O más rápidas. Sin embargo, estos mismos valores en general, también dan lugar a efectos de línea de transmisión en la placa de circuito impreso (PCB).

2.02.2.1 IOBs Organizados en bancos.

La arquitectura de Spartan-3E organiza los IOBs en cuatro bancos. Cada banco mantiene separadas las fuentes VCCO y VREF. Las fuentes separadas permiten a cada banco establecer VCCO independientemente. Del mismo modo, las fuentes de VREF se pueden establecer para cada banco.

Protección ESD. Diodos protegen a todos los pads del dispositivo contra daños ocasionados por descargas electrostáticas (ESD), así como de transitorios de voltajes excesivos.

2.02.3 Bloques de Lógica Configurable (CLB).

Los CLBs constituyen el recurso lógico principal para implementar circuitos lógicos. Cada CLB está compuesto de cuatro slices agrupados en

pares, cada slice contiene dos LUT (Look-Up Tables) para implementar lógica, y dos elementos de almacenamiento dedicados que pueden ser usados como flip-flops o latches. La mayoría de la lógica de uso general en un diseño se asigna automáticamente a los slices en la CLB. Cada CLB es idéntico, están dispuestos en una matriz regular de filas y columnas.

Cada par de slices está organizado como una columna con una cadena independiente de acarreo (CIN). El par izquierdo admite lógica y funciones de memoria y sus slices se llaman SLICEM. El par de la derecha soporta solo lógica y sus slices se llaman SLICEL. Por lo tanto la mitad de los LUTs soportan lógica y pueden ser usados como una memoria de 16x1 (RAM16) o como un registro de corrimiento de 16 bits (SRL16), mientras que la otra mitad solo lógica.

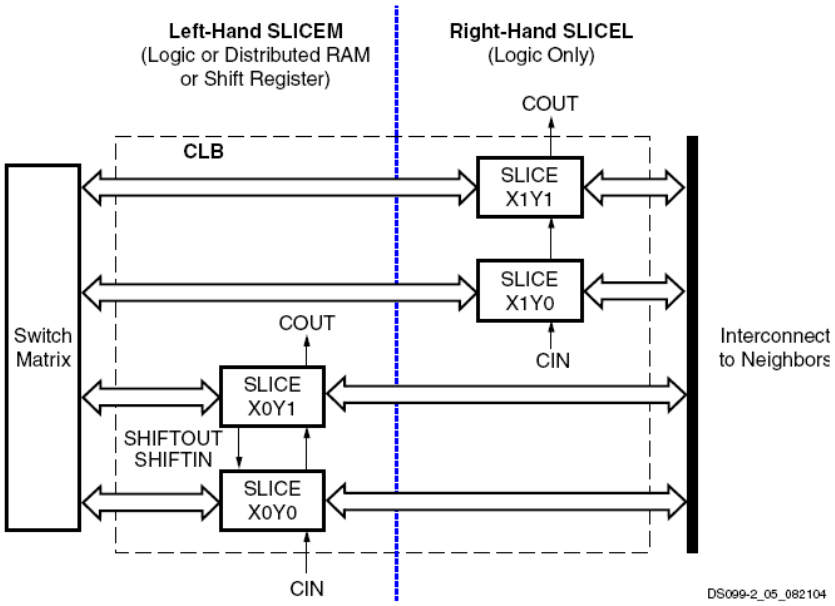

Figura 2.4. Arreglo de slices en un CLB. Imagen tomada de [14], p. 23.

Ambos SLICEM y SLICEL tienen los siguientes elementos en común:

• Dos LUT de 4 entradas, F y G

• Dos elementos de almacenamiento

• Dos multiplexores, F5MUX y FiMUX

• Acarreo y la lógica aritmética

La combinación de un LUT y un elemento de almacenamiento se conoce como "celda lógica". Las características adicionales en un slice, tales como los multiplexores, la lógica de acarreo, y las compuertas aritméticas, hacen que pueda implementar lógica que de otra manera necesitaría LUTs adicionales.

La Figura 2.5 es un diagrama detallado de un SLICEM. Representa un circuito completo de los elementos y conexiones que se encuentran en todos slices. Las líneas punteadas y las azules indican los recursos que se encuentran sólo en el SLICEM y no en el SLICEL.

Cada slice tiene dos mitades, las entradas de control para el reloj (CLK), habilitar reloj (CE), habilitar escritura en el slice (SLICEWE1), y el Reset/Set (RS) son utilizadas por las dos mitades.

El Look-Up Table o LUT es un generador de funciones basado en RAM y es el principal recurso para la implementación de funciones lógicas, en cada SLICEM se puede configurar como memoria RAM distribuida o como registro de corrimiento de 16 bits, lo que permite contar con espacios de memoria de 16 bits en cualquier parte de la topología del FPGA. Los LUTs localizados en las partes superior e inferior del slice se denominan "G" y "F", o "G-LUT" y "F-LUT" respectivamente; tienen cuatro entradas lógicas (A1-A4) y una sola salida (D). Los elementos de almacenamiento en las partes superior e inferior del slice se denominan FFY y FFX, proveen un medio para sincronizar datos a una señal de reloj, entre otros usos. Cualquier operación lógica booleana de cuatro variables se puede implementar en una LUT.

Cada slice tiene dos multiplexores con F5MUX en la parte inferior del slice y FiMUX en la parte superior. Dependiendo del slice, el FiMUX toma el

nombre F6MUX, F7MUX o F8MUX, de acuerdo con su posición en la cadena de multiplexores. La designación indica el número de entradas posibles. Por ejemplo, un F7MUX puede generar cualquier función de siete entradas. Los multiplexores pueden usarse para combinar LUTs dentro del mismo CLB o incluso a través de diferentes CLBs, haciendo posible funciones con mayor número de variables.

La cadena de acarreo entra en la parte inferior del slice como CIN y sale en la parte superior como COUT. Cinco multiplexores controlan la cadena: CYINIT, CY0F y CYMUXF en la parte inferior y CY0G y CYMUXG en la parte superior. La lógica aritmética dedicada incluye las compuertas or exclusivas, XORF y XORG, así como las puertas AND, FAND y GAND. La cadena de acarreo, en combinación con varias compuertas lógicas dedicadas, soporta implementaciones rápidas de operaciones matemáticas.

Los multiplexores de función amplia combinan las LUTs para permitir operaciones lógicas más complejas, cada slice tiene dos de éstos, en la Figura 2.5 corresponden a F5MUX y F1MUX.

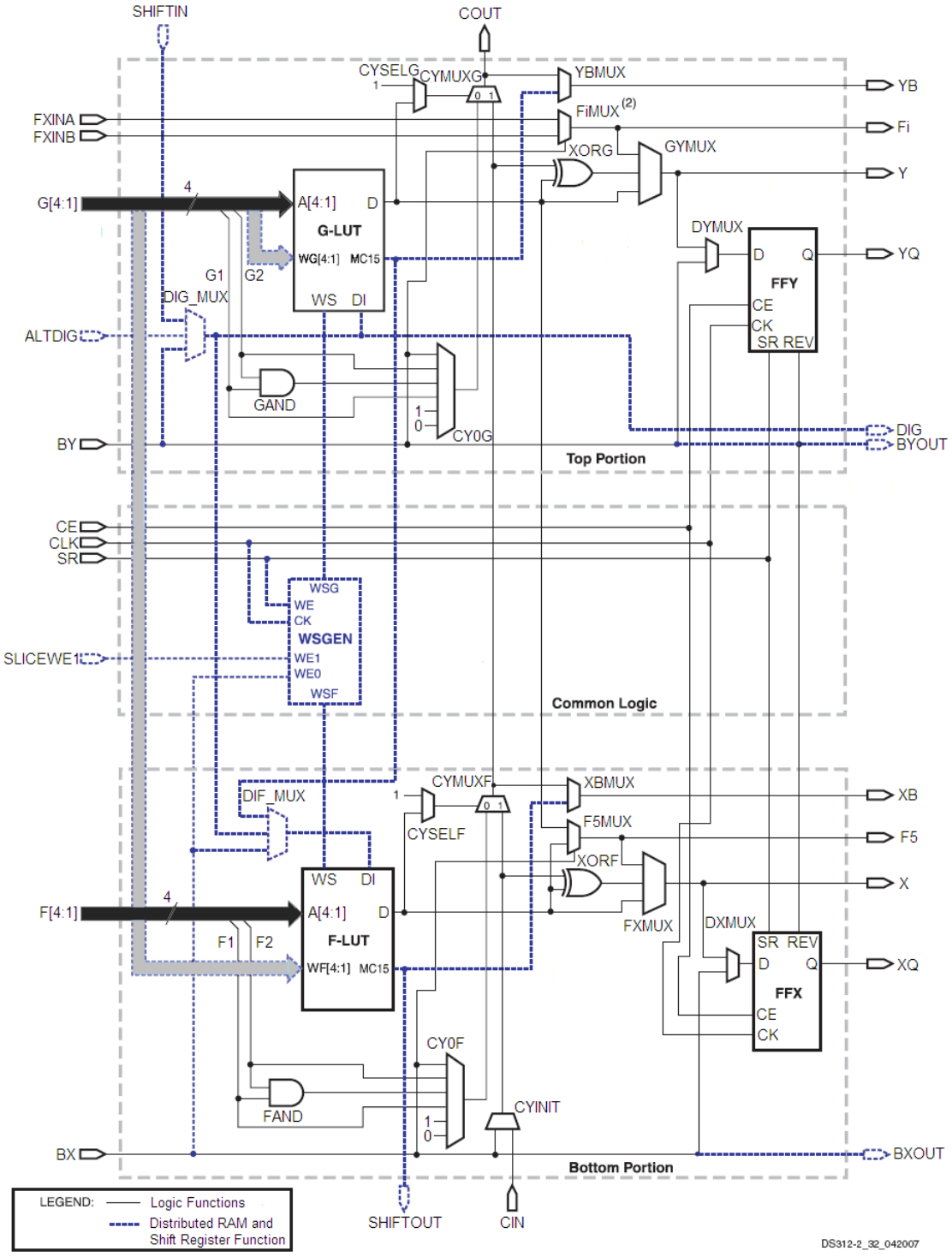

Figura 2.5. Diagrama simplificado de una slice del lado izquierdo de un CLB. Imagen tomada de [14], p. 22.

Hay dos rutas de datos casi idénticas en la parte superior e inferior del slice que son fundamentales para el funcionamiento de cada slice. La ruta básica se origina en la matriz de interruptores de interconexión colocada fuera del CLB. Cuatro líneas, F1 a F4 (o del G1 al G4 en el camino superior), entran en la slice y se conectan directamente a la LUT. Una vez dentro de la slice, los 4 bits de la ruta inferior pasan a través de 'F' una LUT (o "G") que realiza operaciones lógicas. La ruta de salida del LUT, "D", ofrece cinco rutas posibles:

- Salir de la slice por la línea “X” (o “Y”) y volver a interconectarse.
- Dentro de la slice, “X” (o “Y”) sirve como entrada al DXMUX (o DYMUX) que alimenta el dato de entrada, D, correspondiente al elemento de almacenamiento FFX (o FFY). La salida Q de este elemento dirige la ruta XQ (o YQ) que sale del slice.
- Controlar el multiplexor CYMUXF (o CYMUXG) de la cadena de acarreo.
- Con la cadena de acarreo, sirve como una entrada a la compuerta XORF (o XORG), que realiza operaciones aritméticas y produce el resultado en X (o Y).
- Manejar el multiplexor F5MUX para implementar funciones lógicas más anchas que 4 bits. Las salidas D de los F-LUT y G-LUT sirven de entradas de datos para este multiplexor.

En suma a estos caminos lógicos principales, existen dos rutas de bypass que entran a la slice como BX y BY. Una vez dentro del FPGA, BX en la parte de debajo de la slice (o BY en la parte superior) puede tomar varios caminos diferentes:

- Hacer bypass de la LUT y del elemento de almacenamiento, luego salir de la slice como BXOUT (o BYOUT) y volver a interconectarse.

- Hacer bypass a la LUT, y luego pasar a través del elemento de almacenamiento por la entrada D, para luego salir como XQ (o YQ).
- Controlar el multiplexor F5MUX (o FiMUX).
- Servir como una entrada a la cadena de acarreo por medio de los multiplexores.
- Manejar la entrada DI de la LUT.
- BY puede controlar la entrada REV de los elementos de memoria FFY y de FFX.
- Finalmente, el multiplexor DIG_MUX puede conmutar la ruta BY hacia la línea DIG que sale de la slice.

2.02.4 Bloques dedicados de memoria RAM.

La Spartan IIIE tiene de 4 bloques de 36 bloques dedicados de memoria RAM, la cual esta organizada como bloques de 18Kbits de doble puerto configurable. Se puede combinar varios de éstos para formar memorias más anchas o de mayor profundidad.

Los bloques de memoria RAM tienen una estructura de doble puerto. Dos puertos idénticos llamados A y B permiten acceso independiente al mismo bloque de memoria, que tiene una capacidad máxima de 18 432 bits – o 16 384 cuando no se usan los bits de paridad. Cada puerto tiene su propio conjunto de líneas de control, datos y de reloj para operaciones síncronas de lectura y escritura.

Hay cuatro rutas básicas de datos, como se muestra en la Figura 2.6:

1. Escribir y leer del puerto A

2. Escribir y leer del puerto B

3. Transferencia de datos del puerto A al puerto B

4. Transferencia de datos del puerto B al puerto A

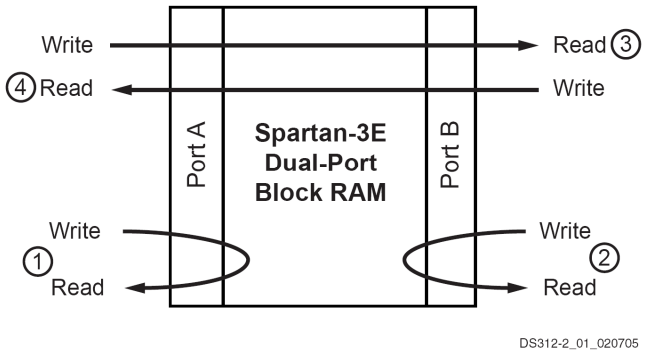

Figura 2.6. Diagrama de un bloque de RAM dedicado de la Spartan III. Imagen tomada de [14], p. 35.

2.02.5 Multiplicadores dedicados.

El Spartan IIIE provee multiplicadores embebidos que aceptan palabras de 18 bits como entrada y entregan productos de 36 bits. Los buses de entrada de estos multiplicadores aceptan datos en complemento dos (tanto 18 bits con signo, como 17 bits sin signo). Adyacentes a cada bloque de memoria RAM está un multiplicador de 18x18. Los 16 bits superiores del bus del puerto A de entrada de datos del bloque de memoria se comparten con los 16 bits superiores del bus de entrada del multiplicando A de el multiplicador. Del mismo modo, los 16 bits superiores del bus del puerto B de entrada de datos se comparten con el bus de entrada del multiplicando B del multiplicador.

Los bloques multiplicadores realizan principalmente una multiplicación numérica de complemento a dos, pero también pueden llevar a cabo algunas aplicaciones menos obvias, como el almacenamiento de datos y corrimientos circulares. Los slice también implementan multiplicadores pequeños y de ese modo completar los multiplicadores dedicados.

Cada multiplicador realiza la operación P = A × B, donde 'A' y 'B' son palabras de 18 bits en complemento a dos, y 'P' es el producto de 36 bits, también en complemento a dos. Las entradas de 18 bits representan valores

que van desde -131,07210 a +131,07110 con un producto que resulta desde -17,179,738,11210 a +17,179,869,18410.

2.02.6 Digital Clock Manager (DCM) y red de distribución de relojes.

El Spartan IIIE tiene 2, 4 o 8 DCM dependiendo del tamaño del dispositivo. Proporcionan un control flexible y completo sobre la frecuencia de reloj, cambio de fase y asimetría de la red de relojes del FPGA. Para lograr esto, el DCM emplea un Delay-Locked Loop (DLL), un sistema de control totalmente digital que utiliza retroalimentación para mantener las características de la señal del reloj con un alto grado de precisión a pesar de las variaciones normales de la temperatura y el voltaje de operación.

El DCM realiza tres funciones principales:

- Eliminación de la asimetría del reloj: El concepto de asimetría describe el grado al cual las señales de reloj pueden, bajo circunstancias normales, desviarse del alineamiento de la fase cero. Ello ocurre cuando pequeñas diferencias en los retardos de las rutas causan que la señal de reloj llegue a diferentes puntos del circuito en tiempos diferentes. El DCM elimina la asimetría por medio de una alineación de fase de la señal de salida del reloj que se genera con la señal de reloj entrante.

- Síntesis de frecuencia: El DCM puede generar diferentes frecuencias de reloj de salida de la señal de reloj entrante. Ello se logra multiplicando y/o dividiendo la frecuencia del reloj de entrada.

- Corrimiento de fase: El DCM puede producir desfases controlados de la señal de reloj de entrada y producir con ello relojes de salida con diferentes fases.

Cada DCM tiene cuatro componentes funcionales relacionados entre si: El Delay-Locked Loop (DLL), El Sintetizador Digital de Frecuencia (DFS), el Desplazador de fase (PS) y lógica de estado.

La Figura 2.7 muestra un diagrama de bloques de este elemento funcional del FPGA.

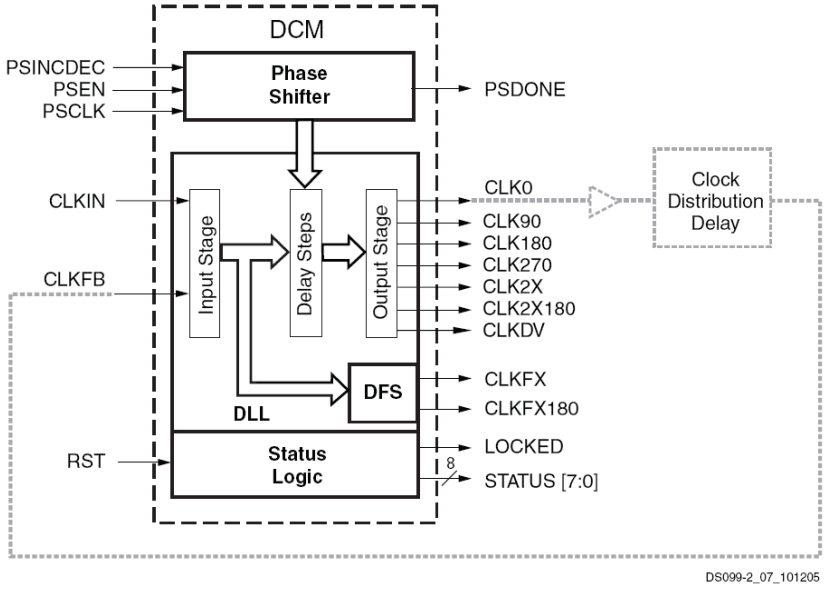

Figura 2.7. Diagrama de bloques de uno de los cuatro DCMs y las señales asociadas. Imagen tomada de [14], p. 48.

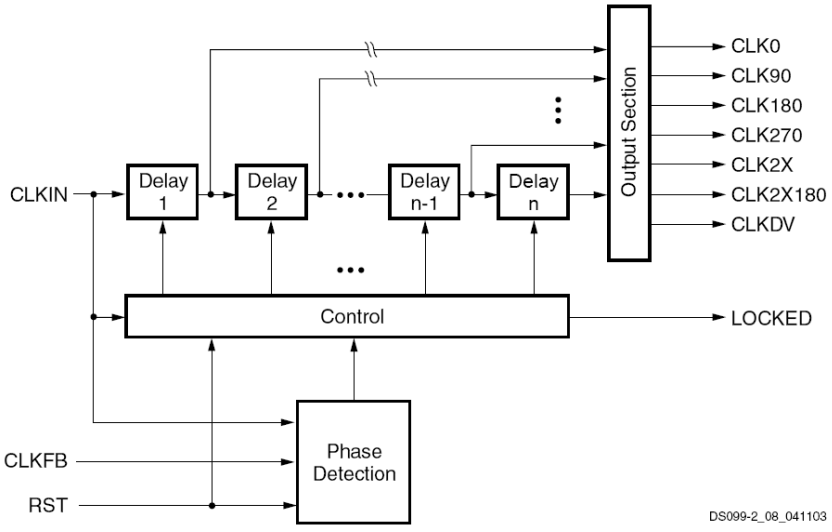

Figura 2.8. Diagrama funcional del Delay-Locked Loop (DLL). Imagen tomada de [14], p. 49.

El DLL tiene como principal función eliminar la asimetría del reloj. La ruta principal del DLL consiste en una etapa de entrada, seguida por una serie de elementos de retardo discreto o taps, los cuales conducen a una etapa de salida. Esta ruta, junto con lógica para detección y control de fase forman un sistema completo con retroalimentación, tal como se muestra en la Figura 2.8.

La señal de reloj aplicada a la entrada CLKIN sirve como una forma de onda de referencia. La DLL tiene por objeto alinear el filo de subida de la señal retroalimentada a la entrada CLKFB con el filo de subida de la entrada CLKIN.

Al eliminar la asimetría del reloj, el enfoque común de utilizar el DLL es el siguiente: La señal de CLK0 se pasa a través de la red de distribución de reloj que alimenta a todos los registros que sincroniza. Estos registros son ya sea interno o externo al FPGA. Luego de pasar por dicha red, la señal de reloj retorna al DLL a través de la entrada CLKFB. El bloque de control del DLL mide el error de fase entre ambas señales, que es una medida de la asimetría del reloj que toda la red introduce. El bloque de control activa el número apropiado de elementos de retardo para cancelar la asimetría de reloj.

La infraestructura de las señales de reloj del Spartan-3E, se muestran en la figura 2.9, provee líneas de interconexión de baja capacitancia, y baja asimetría para transportar señales de alta frecuencia en el FPGA. La infraestructura también incluye las entradas de reloj y los relojes buffers/multiplexores BUFGMUX.

El enrutamiento del reloj dentro del FPGA está basado en cuadrantes, como se muestra en la Figura 2.9. Cada cuadrante soporta ocho señales del reloj, etiquetadas de la 'A' a la 'H'. La fuente de reloj para una línea de reloj individual se origina ya sea desde un elemento global BUFGMUX a lo largo

de los bordes superior e inferior o de un elemento BUFGMUX a lo largo del borde asociado. Las líneas de reloj alimentan a los elementos síncronos (CLBs, IOBs, bloques de memoria RAM, los multiplicadores, y DCM) en el cuadrante.

Esta red de distribución de señales de reloj es completamente independiente de la malla de interconexiones entre CLBs.

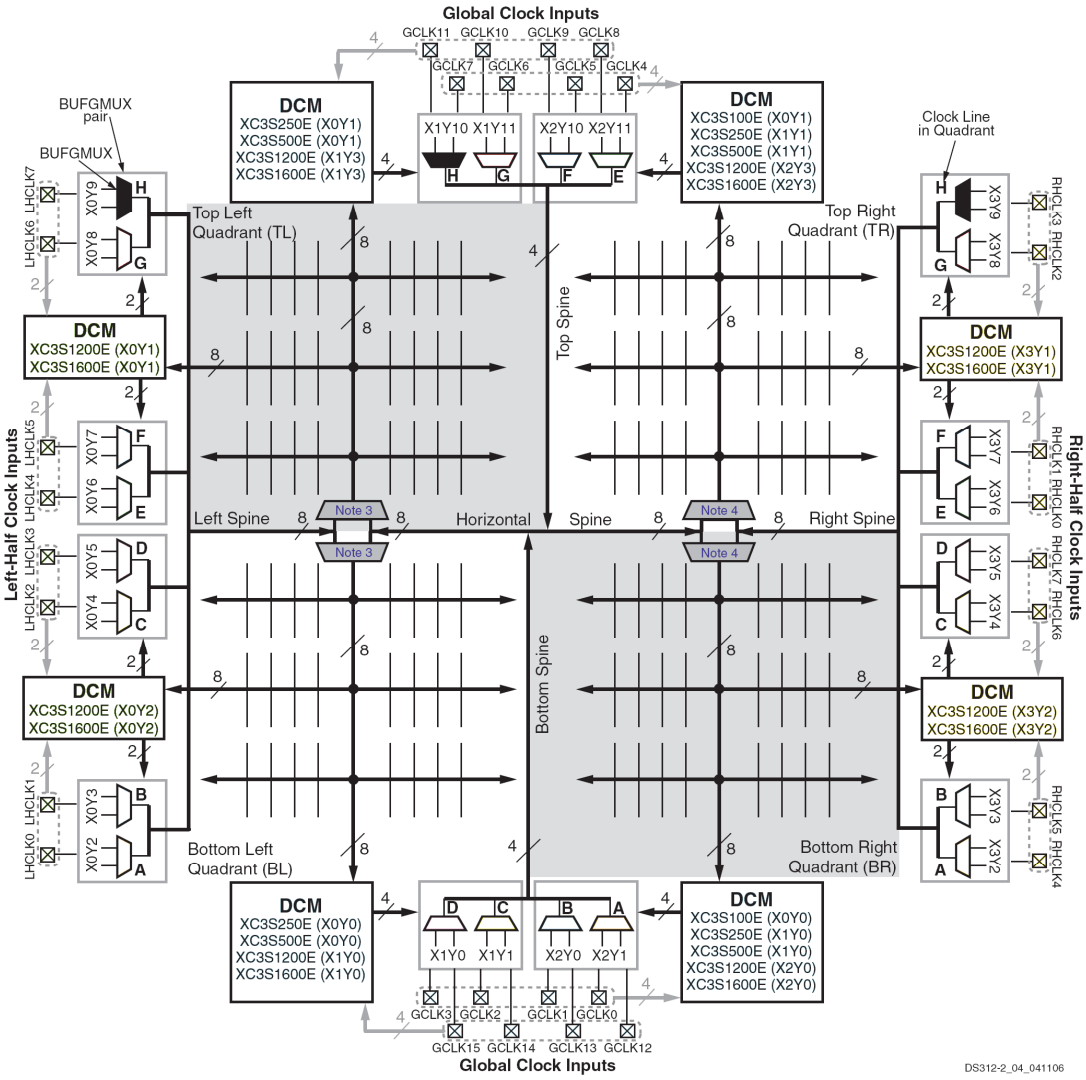

Figura 2.9. Red de distribución de señales de reloj de la Spartan III. Imagen tomada de [14], p. 60.

2.02.7 Red de interconexiones del FPGA.

La red de interconexión es una red programable de rutas de señales entre las entradas y salidas de los elementos funcionales del FPGA (tales como IOB, CLB, DCM y bloques de memoria RAM) Hay cuatro tipos de interconexiones de propósito general disponibles: Long_line, Hex_line, Double_line y Direct_line.

Una matriz de conmutación conecta los diferentes tipos de interconexión a través del FPGA. Un mosaico de interconexión mostrado en la figura 2.10, se define como una sola matriz de conmutación conectada con un elemento funcional, como un CLB, IOB, o DCM. Si un elemento funcional se extiende a través de varias matrices de conmutación, tales como bloques de memoria RAM o multiplicadores.El mosaico de interconexión se define por el número de matrices de conmutación conectadas a ese elemento funcional. Un dispositivo Spartan-3E se puede representar como una matriz de mosaicos de interconexión donde los recursos de interconexión son para el canal entre dos mosaicos de interconexión adyacentes ya sea de filas o columnas.

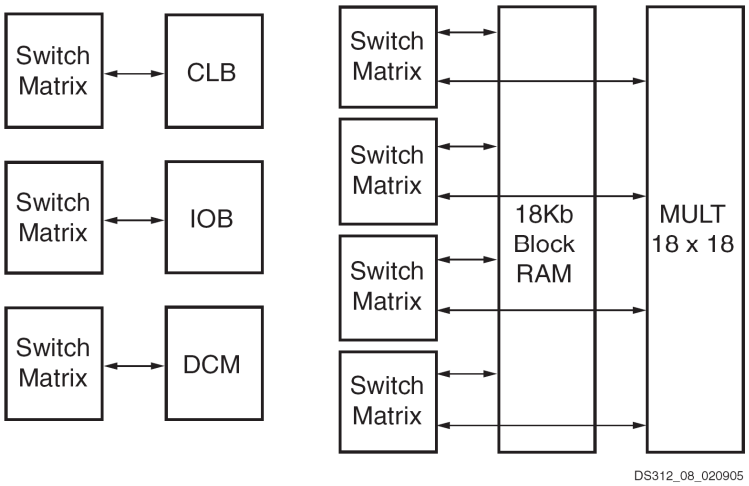

Figura 2.10. Tipos de mosaicos de interconexión CLB, IOB, DCM y bloques de memoria RAM y multiplicadores. Imagen tomada de [14], p. 64.

2.02.7.1 Interconexiones de propósito general.

Long_line. Cada conjunto de 24 señales long_line se extiende tanto horizontal como verticalmente y se conecta a uno de cada seis mosaicos de interconexión. En cualquier mosaico, cuatro de las interconexiones long_line transportan o reciben señales de una matriz de conmutación. Debido a su baja capacitancia, estas líneas están bien adaptadas para llevar señales de alta frecuencia con mínimos efectos de carga. Si todas las líneas de reloj ya se han utilizado y señales adicionales de reloj aún no se han asignado, las filas largas son una buena alternativa.

Hex_lines. Cada grupo de ocho señales hex_lines está conectado a uno de cada tres mosaicos de interconexión, tanto horizontal como verticalmente. Treinta y dos líneas hex_line están disponibles entre cualquier mosaico de interconexión.

Double_line. Cada conjunto de ocho líneas double_line están conectados intercaladamente entre mosaicos de interconexión, tanto horizontal como verticalmente. En las cuatro direcciones. Treinta y dos líneas double_line están disponibles entre cualquier mosaico de interconexión.

Direct_line. Entregan conexiones directas de cada CLB hacia cada uno de sus ocho vecinos (2.11d). Estas líneas son usadas más a menudo para conducir una señal proveniente de un CLB de origen hacia una Double_line, Hex_line o Long_line y desde esa ruta larga hacia otra Direct_line que llevará la señal hacia el CLB de destino.

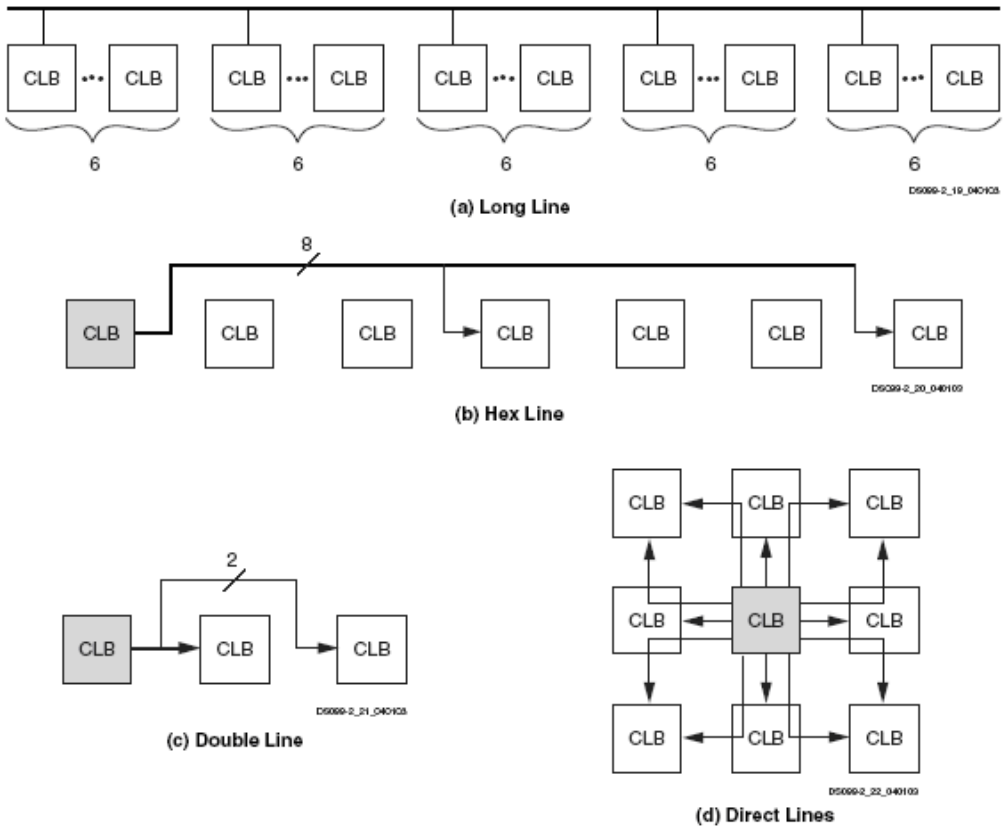

Figura 2.11. Tipos de interconexiones entre CLBs en la Spartan III.

2.02.8 Proceso de configuración del FPGA Spartan III.

El FPGA Spartan IIIE se programa por medio de la carga de los datos de configuración en celdas de memoria estática, las que colectivamente controlan todos los elementos funcionales y los recursos de interconexión. Luego de aplicar alimentación, se escribe la trama de configuración en dicha memoria utilizando uno de los siguientes modos: Maestro - Paralelo, Esclavo - Paralelo, Maestro - Serial, Esclavo - Serial o Boundary-Scan (JTAG). Estos modos difieren en el origen del reloj (proviene del FPGA en los modos Maestro y es externo en los modos Esclavo), y en la forma en que se escriben los datos, por lo que los modos paralelos son más rápidos.

El modo Boundary-Scan utiliza pines dedicados del FPGA y cumple con los estándares IEEE 1149.1 Test Access Port e IEEE 1532 para

dispositivos In-System Configurable (ISC). Este modo está siempre disponible en el FPGA y al activarlo se desactivan los otros modos ya mencionados.

El proceso de configuración del FPGA ocurre en tres etapas. Primero la memoria interna de configuración es borrada. Luego los datos de configuración son cargados en dicha memoria, y finalmente la lógica es activada por un proceso de partida.

2.02.9 Flujo de diseño para la configuración del FPGA.

El flujo de diseño para generar la configuración de un FPGA está compuesto principalmente por cuatro etapas: diseño lógico, síntesis, implementación y generación del archivo de salida. En el caso de los FPGA de Xilinx existe un paquete de software que reúne herramientas para llevar a cabo cada una de estas etapas, subetapas y procesos de simulación en varios niveles de profundidad. Esta utilidad se llama Xilinx Integrated Software Environment (ISE) y está disponible, en una versión bastante completa y gratuita, con el nombre de WebPack ISE

El diseño lógico se realiza mediante un lenguaje de descripción de hardware tal como VHDL o Verilog. En este trabajo se ha utilizado el primero para describir cada uno de los módulos del diseño, y también se ha utilizado una herramienta de más alto nivel, que de manera gráfica permite juntar los distintos bloques en un esquemático y unirlos con buses y conexiones unitarias. Esta herramienta es parte del WebPack ISE y se vale de la característica jerárquica del mismo VHDL, mediante la cual se puede crear componentes y unirlos usando recursos del lenguaje.

Una vez descrito el sistema, la siguiente etapa consiste en sintetizarlo. Este proceso se realiza de forma automática y sigue directivas de configuración, en las que se determinan los algoritmos preferidos de síntesis.

La salida de éste es una netlist, que es un archivo que contiene una lista de conexiones, una lista de instancias y para cada instancia, una lista de señales conectadas a los terminales de dicha instancia. Además contiene información de atributos del diseño. En este caso la netlist es una descripción a nivel de compuertas lógicas del sistema descrito. La herramienta que realiza este proceso en el caso del paquete ISE se llama Xilinx Synthesis Tool (XST) y su netlist es un archivo de formato NGC (Native Generic Circuit).

La netlist de formato NGC es la entrada para el proceso de implementación, el cual se subdivide en tres etapas: Translate, Mapping y Place and Route. En la primera el archivo NGC es convertido a un formato estándar llamado NGD, por una herramienta llamada NGDBuild, que no sólo acepta archivos de salida del sintetizador XST, si no que también otros formatos provenientes de otros sintetizadores alternativos. En la segunda etapa se mapea el diseño lógico contenido en el archivo NGD, en los componentes físicos reales, con que cuentan los slices del FPGA. En la tercera, se determina la topología de colocación y de interconexión de los elementos ya mapeados. Esta etapa es un proceso iterativo, que tiene alto costo computacional y puede demorar decenas de minutos.

Finalmente, en la etapa de generación, otra herramienta genera un archivo de configuración, el que es descargado a la memoria del FPGA y que contiene la trama de bits que produce la configuración adecuada.

2.03 Introducción a VHDL.

Un lenguaje de descripción de hardware (HDL por sus siglas en ingles) es similar a un lenguaje de programación de computadora, excepto que un HDL es usado para describir hardware. Dos HDLs son estándares de la IEEE: el VHDL que significa VHSIC (Very High Speed Integrated Circuits)

Hardware Description Language y verilog HDL, ambos presentan características muy similares.

VHDL provee de portabilidad de diseño. Un circuito que se especifica en VHDL puede ser implementado en un dispositivo lógico programable o en un circuito integrado sin cambiar la especificación VHDL utilizando las herramientas CAD proporcionadas por las diferentes compañías.

Un circuito lógico se hace escribiendo código VHDL. Las señales en el circuito pueden ser representadas como variables en el código fuente, y las funciones lógicas se expresan mediante la asignación de valores a estas variables.

El código fuente VHDL es texto sin formato. Similar a la forma en la que los circuitos grandes se manejan en una captura esquemática, el código VHDL se puede escribir en forma modular lo que facilita el diseño jerárquico; se puede dividir un sistema complicado en subsistemas más sencillos, tantas veces como sea necesario hasta poder resolver cada módulo (subsistema) por separado. Ello facilita la prueba de cada módulo independientemente y da más seguridad al correcto funcionamiento del sistema final. Diseños de circuitos lógicos pequeños y grandes se pueden representar de manera eficiente en código VHDL.

La síntesis es el proceso de generar un circuito lógico desde una especificación inicial que puede ser dada en la forma de un diagrama esquemático o código escrito en un lenguaje de descripción de hardware.

Por lo general, la asignación es de descripciones abstractas a descripciones más detalladas, más cerca de la forma definitiva para su aplicación. Por ejemplo, una descripción VHDL podría ser asignada a un conjunto de ecuaciones booleanas que preserven el comportamiento de la especificación original. Una segunda herramienta puede tomar estas

ecuaciones booleanas y una biblioteca de compuertas disponibles en una determinada tecnología, y generar una descripción a nivel de compuertas del sistema

El proceso de traducción o compilación, del código VHDL en una red de compuertas lógicas es parte de la síntesis. La salida es un conjunto de expresiones lógicas que describen las funciones lógicas necesaria para realizar el circuito. Un circuito representado en la forma de expresiones lógicas puede ser simulado para verificar que su funcionamiento será el esperado. Los simuladores son programas que pueden ejecutar dinámicamente una descripción abstracta del diseño.

Dada la descripción del circuito y un modelo de cómo los elementos de la descripción se comportan, el simulador mapea un estímulo de entrada en una respuesta de salida, a menudo en función del tiempo. Si el comportamiento no es el esperado, en la mayoría de lo casos es más fácil identificar y reparar el problema en esta parte del diseño que resolver los problemas en el producto final. Los simuladores existen para todos los niveles de descripción del diseño, desde el nivel de comportamiento más abstracto hasta el nivel de transistor.

Para nuestros propósitos, dos formas de simulación son los más relevantes: la lógica y la de tiempos. La simulación lógica modela el diseño como compuertas lógicas interconectadas, se utiliza el simulador para determinar si el comportamiento de la tabla de verdad del circuito cumple las expectativas. La simulación de tiempos es como la simulación lógica, excepto que introduce retrasos. La simulación no solo calcula las salidas en base a las entradas, también toma en cuenta el tiempo de retraso de las señales.

Después de la síntesis el siguiente paso en el flujo de diseño es determinar exactamente cómo implementar el circuito en un chip determinado. Un software *place-and-route* genera la distribución física para el

FPGA. Este software mapea un circuito especificado en la forma de expresión lógica en un circuito real que hace uso de los recursos disponibles en el chip. Determina la colocación de elementos de lógica específica, así como las conexiones del cableado que tienen que ser hechas entre estos elementos para implementar el circuito deseado.

Existen dos formas de describir un circuito. Por un lado se puede describir un circuito indicando los diferentes componentes que lo forman y su interconexión, de esta manera se tiene especificado un circuito y se sabe como funciona. La segunda forma consiste en describir un circuito indicando lo que hace o cómo funciona, es decir, describiendo su comportamiento. Naturalmente esta forma de describir un circuito es mejor para un diseñador puesto que lo que realmente le interesa es el funcionamiento del circuito más que sus componentes.

La sintaxis de VHDL no es sensible a mayúsculas o minúsculas, por lo que se puede escribir como se prefiera y es libre de formato; espacios y líneas en blanco se pueden insertar libremente.

Un bloque independiente de código VHDL esta compuesto de al menos tres secciones fundamentales.

Declaración de las librerías (LIBRARY). Una librería es una colección de bloques de código usados comúnmente. Colocando tales bloques en una librería les permite ser usadas o compartidas por otros diseños. Se necesitan dos líneas de código, una contiene el nombre de la librería y la otra la cláusula use.

```
Library ieee;
use eee.std_logic_1164.all;
```

Llaman al paquete std_logic_1164 de la librería de la ieee. El paquete y la librería permiten añadir tipos adicionales, operadores, funciones, etc. a VHDL.

Declaración de la entidad (ENTITY). Especifica los pines de entradas y salidas del circuito. Es una lista con las especificaciones de todos los pines de entradas y salidas (puertos) del circuito. El nombre de la entidad puede ser cualquier nombre, excepto palabras reservadas. La entidad únicamente describe la forma externa del circuito, es análoga a un símbolo esquemático de los diagramas electrónicos, el cuall describe las conexiones del dispositivo hacia el resto del diseño.

Ejemplo de una entidad:

```
ENTITY mux IS
PORT (a:     IN bit;
      b:     IN bit;
      Selec:       IN bit;
      Salida:      OUT bit);
End mux;
```

La primera línea indica el nombre de la entidad (mux), las entradas y salidas se denominan puertos (ports en ingles) y son declarados en la sección PORT, cada puerto tiene un modo asociado que especifica sii es entrada (IN) o salida (OUT). Cada puerto representa una señal, por lo tanto tiene un tipo de señal asociado; el modo de una señal puede ser: IN, OUT, INOUT, o BUFFER y el tipo de señal puede ser BIT, STD_LOGIC, INTEGER, etc.

Este ejemplo tiene tres entradas (modo IN) y una salida (modo OUT) de tipo bit.

Declaración de la arquitectura (ARCHITECTURE). Contiene propiamente el código VHDL que describe cómo se debe comportar el circuito.

La entidad especifica las entradas y salidas para el circuito, pero no da detalles de lo que el circuito representa. El bloque de arquitectura, es dónde se describe el circuito, puede ser una descripción estructural o comportamental, ambas son descripciones diferentes pero corresponden al mismo circuito, símbolo o entidad. VHDL permite múltiples bloques de arquitectura en una identidad, cuando se compile se indica cuál es la arquitectura que se quiere utilizar. El bloque de arquitectura tiene dos partes: una parte declarativa, donde señales y constantes son declaradas, y la parte del código.

Sintaxis:

```
ARCHITECTURE arch_name OF entity_name IS
        -- declaraciones de la arquitectura
        -- tipos
        -- señales
        -- componentes
BEGIN
        -- código de descripción
        -- instrucciones concurrentes
        -- ecuaciones booleanes
        END arch_name;
```

Ejemplo:

```
ARCHITECTURE comportamental OF mux IS
BEGIN
PROCESS(a,b,selec)
BEGIN
        IF (SELEC='0') THEN
                Salida <= a;
        ELSE
                Salida <= b;
        END IF;
END PROCESS;
END comportamental;
```

Esta descripción comportamental sigue una estructura parecida a los lenguajes de programación convencionales. Más que especificar la estructura

o la forma en que se deben conectar los componentes de un diseño, nos limitamos a describir su comportamiento. Una descripción comportamental consta de una serie de instrucciones, que ejecutadas modelan el comportamiento del circuito. Esta forma de describir el circuito permite a ciertas herramientas sintetizar el diseño. La diferencia con un netlist es que no se están indicando ni los componentes ni sus interconexiones, sino simplemente lo que hace, es decir, su comportamiento o funcionamiento.

La arquitectura del ejemplo del mux utiliza el operador de asignación de señal “<=” para asignar el valor correspondiente al puerto de salida, si la señal selec es cero, entonces la salida es la entrada a, y si selec es uno, es la entrada b.

VHDL posee una forma de describir circuitos que además permite la paralelización de instrucciones, y que se encuentra más cercana a una descripción estructural del mismo, siendo todavía una descripción funcional. A continuación se muestran dos ejemplos de una descripción concurrente.

```
ARCHITECTURE flujo1 OF mux  IS
SIGNAL nosel, ax, bx: bit;
BEGIN
        nosel <= NOT selec;
        ax <= a AND nosel;
        bx <= b AND selec;
        salida <= ax OR bx;
END flujo1;
```

```
ARCHITECURE flujo2 OF mux IS
BEGIN
      salida <= a WHEN selec = ‘0’
ELSE  b;
END flujo2;
```

En la descripción de la izquierda hay varias instrucciones todas ellas concurrentes, es decir, se ejecutan cada vez que cambia alguna de las señales que intervienen en la asignación. Este primer caso es casii una descripción estructural, ya que de alguna manera se están describiendo las señales (cables) y los componentes que la definen; aunque no es estructural, ya que en realidad se trata de asignaciones a señales y no una lista de

componentes y conexiones. Las instrucciones concurrentes son como partes del circuito que operan en paralelo.

2.03.1 Elementos sintácticos.

VHDL es un lenguaje, por lo que tiene sus elementos sintácticos, sus tipos de datos y sus estructuras como cualquier otro tipo de lenguaje. El hecho de que sirva para la descripción hardware lo hace un poco diferente de un lenguaje convencional. Una de estas diferencias es probablemente la posibilidad de ejecutar instrucciones a la vez de forma concurrente.

Comentarios: cualquier línea que empieza por dos guiones “--" es un comentario. El texto después de -- es ignorado.

Identificadores: es lo que se usa para dar nombre a los diferentes objetos del lenguaje como variables, señales, nombres de rutina, etc. puede ser cualquier nombre compuesto por letras y números, incluyendo el símbolo de subrayado “_”. Nunca puede contener ninguno de los símbolos especiales ni puede empezar por un número o subrayado; tampoco se permite que el identificador acabe con un subrayado nii que haya dos seguidos. Por ultimo, no debe haber ningún identificador que coincida con alguna de las palabras clave del VHDL. Las mayúsculas y minúsculas son consideradas iguales.

Números: cualquier número se considera que se encuentra en base 10. Se admite la notación científica convencional para números en punto flotante. Es posible poner números en otras bases utilizando el símbolo de numero “#”.

Caracteres: es cualquier letra o carácter entre comillas simples: ‘1’, ‘3’, ‘t’.

Cadenas: son un conjunto de caracteres englobados por comillas dobles: “Esto es una cadena”.

Palabras reservadas: las palabras reservadas en VHDL, o palabras clave, son aquellas que tienen un significado especial. Son las instrucciones, órdenes y elementos que permiten definir sentencias. Por esta razón, no se deben utilizar como identificadores, ya que tienen un significado diferente.

Concatenación: “&” concatena matrices de manera que la dimensión de la matriz resultante es la suma de las dimensiones de las matrices sobre las que opera: por ejemplo, punto <=x&y constituye el arreglo punto con el arreglo “x” en las primeras posiciones, y el arreglo “y” en las ultimas.

2.03.2 Tipos de datos.

La sintaxis del VHDL es estricta con respecto a los tipos. Cualquier objeto en VHDL debe tener un tipo y sólo los valores y operaciones definidas para ese tipo se pueden aplicar al objeto. En VHDL no existen tipos propios del lenguaje como pueden ser el tipo real o integer (entero), lo que tiene son los mecanismos para poder definir cualquier tipo incluidos éstos.

Cuando se compila el programa se carga una parte de código previa que se encuentra en una biblioteca. Esta parte de código es común a todas las herramientas de VHDL y contiene una serie de definiciones de tipos y funciones que, al ser comunes a todas las herramientas, compiladores, simuladores, etc. casii parecen formar parte del propio lenguaje, pero no es así. De esta forma, por ejemplo, existe un tipo que se llama precisamente integer, pero este tipo no es propio del lenguaje sino que se carga al inicio junto con otros tipos predefinidos. A continuación se muestran las posibles declaraciones de tipos que se pueden hacer y se presenta cómo están especificados estos tipos predefinidos.

Constantes. Los objetos de esta clase tienen un valor inicial que es asignado de forma previa a la simulación y que no puede ser modificado durante ésta.

CONSTANT identificador: tipo:= valor;

Variables. Los objetos de esta clase contienen un único valor que puede ser cambiado durante la simulación con una sentencia de asignación. Las variables generalmente se utilizan como índices, principalmente en instrucciones de bucle, o para tomar valores que permitan modelar componentes. Las variables NO representan conexiones o estados de memoria.

VARIABLE identificador: tipo [:= valor];

Señales. Los objetos de esta clase contienen una lista de valores que incluye el valor actual y un conjunto de valores futuros. Las señales representan elementos de memoria o conexiones y si pueden ser sintetizadas. Los puertos de una entidad son implícitamente declarados como señales en el momento de la declaración, ya que estos representan conexiones. También pueden ser declaradas en la arquitectura antes del BEGIN, lo cual nos permite realizar conexiones entre diferentes módulos.

SIGNAL identificador: tipo;

Tipo **enumerado** es un tipo de dato con un grupo de posibles valores asignados por el usuario. Los tipos enumerados se utilizan principalmente en el diseño de máquinas de estados

TYPE nombre IS (valor1, valor2, …);

Los tipos enumerados se ordenan de acuerdo a sus valores. Los programas de síntesis automáticamente codifican binariamente los valores del tipo enumerado para que estos puedan ser sintetizados. Algunos programas lo hacen mediante una secuencia binaria ascendente, otros buscan cual es la codificación que mejor conviene para tratar de minimizar el circuito o para incrementar la velocidad del mismo una vez que la descripción

ha sido sintetizada. También es posible asignar el tipo de codificación mediante directivas propias de la herramienta de síntesis.

Tipos **compuestos** un tipo compuesto es un tipo de dato formado con elementos de otros tipos, existen dos formas de tipos compuestos, ARRAYS y RECORDS.

ARRAY (arreglos). Los arreglos son una colección de objetos del mismo tipo. Pueden ser de una dimensión (1D), dos dimensiones (2D), o de una dimensión por una dimensión (1Dx1D). Los arreglos pueden ser de dimensiones más grandes pero por lo general no son sintetizables.

Sintaxis:

TYPE nombre_tipo IS *ARRAY* (especificacion) OF tipo_dato;
SIGNAL nombre_señal: nombre_tipo;

RECORD es un objeto de datos que consiste en una “colección” de elementos de distintos tipos.

Sintaxis:

TYPE nombre IS *RECORD*
elemento1: tipo_de_dato1;
elemento2: tipo_de_dato2;
END *RECORD*;

2.03.2.1 Paquetes de datos predefinidos.

VHDL contiene una serie de tipos de datos predefinidos, especificados a través de los estándares IEEE 1076 e IEEE 1164.

Paquete standard de la librería std: define los tipos de datos BIT, BOOLEAN, INTEGER y REAL.

Paquete std_logic_1164, especifica los sistemas lógicos STD_LOGIC (de 8 niveles) y STD_ULOGIC (de 9 niveles).

STD_LOGIC, este tipo representa una lógica multivaluada de 9 valores. Además del '0' lógico y el '1' lógico, posee alta impedancia 'Z', desconocido 'X' ó sin inicializar 'U' se encuentran en simulación, entre otros. Una señal en un circuito digital frecuentemente contiene múltiples bits. El tipo de dato std_logic_vector, se define como un arreglo de elementos std_logic. Por ejemplo:

a: in std_logic_vector (7 downto 0);

Declara un puerto de entrada "a" de 8 bits.

Se puede especificar un rango a(7 downto 5) o un solo termino a(3) para accesar a los elementos de un arreglo.

STD_LOGIC son un subtipo de STD_ULOGIC. Los últimos incluyen un valor lógico extra 'U' (unresolved).

Paquete std_logic_arith: especifica los tipos de datos SIGNED y UNSIGNED y sus operaciones aritméticas y de comparación. También contiene varias funciones de conversión de datos.

Paquete std_logic_signed: contiene funciones que permiten operaciones con datos del tipo STD_LOGIC_VECTOR como si fueran del tipo SIGNED.

Paquete std_logic_unsigned: contiene funciones que permiten operaciones con datos del tipo STD_LOGIC_VECTOR como si fueran del tipo UNSIGNED.

2.03.3 Operadores.

VHDL proporciona varios tipos de operadores predefinidos:

Operadores de asignación. Se utilizan para asignar valores a las señales, variables y constantes.

<= para asignar valores a una señal
:= se usa para asignar valores a una VARIABLE, CONSTANT, o GENERIC.
=> se usa para signar valores a elementos individuales de vectores o con la instrucción OTHERS

Operadores lógicos. Se utilizan para realizar operaciones lógicas, los tipos de datos deben ser: BIT, STD_LOGIC o STD_ULOGIC.

Las operaciones son: NOT, AND, OR, NAND, NOR, XOR, XNOR.

Operadores aritméticos. Se usan para realizar operaciones aritméticas. Los datos pueden ser de tipo: INTEGER, SIGNED, UNSIGNED, o REAL.

Las operaciones son: suma “+”, resta “-“, multiplicación “*”, división “/”, elevación a potencia “**”, modulo “MOD”, residuo “REM”, valor absoluto “ABS”. No hay restricciones en la síntesis de la suma, la resta y la multiplicación. Para división, solo divisiones entre potencias de dos son permitidas, para la exponenciación, solo valores estáticos para la base y el exponente son aceptados. Los últimos tres operadores generalmente tienen poco o nada de soporte en la síntesis.

Operadores de comparación. Se utilizan para hacer comparaciones, pueden ser de cualquier tipo de dato:

“=” igual a,
“/=” diferente a,
“<” menor que,
“>” mayor que,
“<=” menor o igual que,
“>=” mayor o igual que.

Operadores de corrimiento. Usados para corrimiento de datos, son:

SLL – corrimiento a la izquierda, las posiciones de la derecha se llenan con ceros.
SRL – corrimiento a la derecha, las posiciones a la izquierda se llenan con ceros.
SLA – corrimiento a la izquierda, el bit más a la derecha se copia en las posiciones de la derecha.
SRA – corrimiento a la derecha, el bit más a la izquierda se copia en las posiciones de la izquierda.
ROL – rota a la izquierda.
ROR – rota a la derecha.

Operadores de concatenación. Se utilizan para agrupar valores, los datos pueden ser de cualquier tipo utilizados en operaciones lógicas. Los operadores de concatenación son: &, (,,,)

2.03.4 Genéricos.

GENERIC (genéricos) es una forma de especificar parámetros genéricos, es un parámetro estático que puede ser modificado y adaptado fácilmente a diferentes aplicaciones. Una sentencia genérica debe ser declarada en la entidad. El parámetro así especificado será global. Su sintaxis es:

GENERIC (nombre _ parámetro: tipo _ parámetro := valor _ parámetro)

2.03.5 Código secuencial.

Procesos (*PROCESS*). Un proceso es una sección secuencial de código VHDL. Se caracteriza por la presencia de IF, WAIT, CASE o LOOP, y una lista sensible. Un proceso debe estar en el código principall y es ejecutado cada vez que una señal de su lista sensible cambia. Su sintaxis es la siguiente:

```
[etiqueta] PROCESS (lista sensible)
        [VARIABLE nombre_variable TYPE [rango] [:= valor_inicial;]]
BEGIN
        (código secuencial)
END PROCESS [etiqueta];
```

El valor inicial de las variables no es sintetizable.

VHDL tiene dos formas de pasar valores no estáticos: a través de una señal o por medio de una variable. Una señal puede ser declarada en un paquete, entidad o arquitectura, mientras que una variable solo se puede declarar dentro de un bloque de código secuencial.

El valor de una variable no puede salir del proceso directamente, debe ser asignado a una señal. Por otro lado, la actualización de una variable es inmediata, se puede contar con su nuevo valor en la siguiente línea del código. Ese no es el caso de las señales (cuando son usadas en un proceso), su nuevo valor por lo general sólo se garantiza que estará disponible después de la conclusión de la ejecución actual del proceso.

Declaraciones destinadas a código secuencial: IF, WAIT, CASE y LOOP, solo pueden ser usadas dentro de un proceso, función o procedimiento.

IF. La sintaxis de IF es:

```
IF condición THEN
        asignaciones;
ELSIF condición_2 THEN
        asignaciones_2;
ELSE asignaciones_3;
END IF;
```

WAIT. Cuando se utiliza WAIT, el proceso no puede tener una lista sensible. Tiene tres diferentes sintaxis:

1. WAIT UNTIL condición;

Esta sintaxis acepta solo una señal, WAIT UNTIL debe ser la primera línea en el proceso. El proceso se ejecuta cada vez que la condición se cumple.

2. WAIT ON señal1 [, señal2, …]

Esta sintaxis acepta múltiples señales, el proceso se detiene hasta que cualquier señal de su lista sensible cambia.

3. WAIT FOR

Esta sintaxis esta destinada solo para simulación.

CASE. La sintaxis de CASE es:

```
CASE identificador IS
        WHEN valor => asignaciones;
        WHEN valor => asignaciones;
        …
END CASE;
```

Todos los posibles valores de “identificador” deben ser probados, por lo que la palabra clave OTHERS es a menudo útil. La palabra clave NULL se utiliza cuando no se realiza acción alguna.

LOOP. Es útil cuando una parte del código debe ser escrito varias veces.

Tiene varias sintaxis:

FOR/LOOP: El bucle se repite un número fijo de veces.

```
[etiqueta] FOR identificador IN rango LOOP
        (código secuencial)
END LOOP [etiqueta];
```

WHILE/LOOP: El bucle se repite hasta que la condición ya no se cumple.

```
[etiqueta] WHILE condición LOOP
        (código secuencial)
```

END LOOP [etiqueta];

EXIT: se usa para terminar la ejecución de un bucle.

[etiqueta] EXIT [etiqueta] [WHEN condición];

NEXT: se utiliza para saltarse pasos de un bucle.

[etiqueta] NEXT [etiqueta de bucle] [WHEN condición];

2.03.6 Descripción Estructural.

Un sistema digital se compone frecuentemente de varios subsistemas más pequeños. Esto permite construir un sistema complejo de componentes más simples o prediseñados. Esta descripción utiliza entidades descritas y compiladas previamente. Se declaran los componentes que se utilizan y después, mediante los nombres de los nodos, se realizan las conexiones entre las compuertas. Las descripciones estructurales son útiles cuando se trata de diseños jerárquicos.

Una descripción estructural: Describe las interconexiones entre distintos módulos. Estos módulos pueden a su vez tener un modelo estructural o de comportamiento.

Sintaxis de descripción estructural:

```
ENTITY nombre_entidad IS
        port(…);
END entity_name;

ARCHITECTURE nombre_arquitectura OF nombre_entidad IS

COMPONENT nombre_componente
        port (…);
END COMPONENT;
```

```
declaración de señales;
BEGIN
component_i: nombre_componente
        PORT MAP (io_name)
END nombre_arquitectura;
```

Capitulo 3.

Morfología matemática

3.01 Morfología matemática.

La morfología matemática se puede usar como una herramienta para extraer componentes de la imagen que son útiles en la representación y descripción de la forma de ciertas regiones de la imagen.

La teoría de conjuntos es la base para definir las operaciones en morfología matemática. Los conjuntos representan objetos en una imagen, son números enteros en el espacio 2-D (Z^2).

Una transformación morfológica viene dada mediante la relación de una imagen (modelada como un conjunto de puntos A) con un elemento estructurante (un pequeño conjunto de puntos B) expresado respecto a un origen relativo 0, el cual puede ser visto como una sonda que se desplaza sistemáticamente sobre la imagen A, y que escanea y modifica la imagen de acuerdo a una regla específica.

El concepto es bastante simple, una pequeña máscara de tamaño impar de tamaño LxL, se explora sobre una imagen binaria. Si el patrón de valor binario de la máscara coincide con el estado de los píxeles bajo la máscara, un píxel de salida en correspondencia espacial con el píxel central de la máscara se ajusta a un estado binario deseado. Para una falta de coincidencia, el píxel de salida se establece en el estado binario opuesto.

3.01.1 Conceptos básicos de teoría de conjuntos.

Considerar una imagen binaria definida por la función A(j,k), un píxel en la coordenada (j,k) es un miembro de A(j,k), según lo indicado por el símbolo $\in$ si y sólo si es un 1 lógico. Una imagen binaria B(j,k) es un subconjunto de la imagen A(j,k), como se indica por $B(j,k) \subseteq A(j,k)$ si para cada ocurrencia espacial de un 1 lógico de B, A es un 1 lógico.

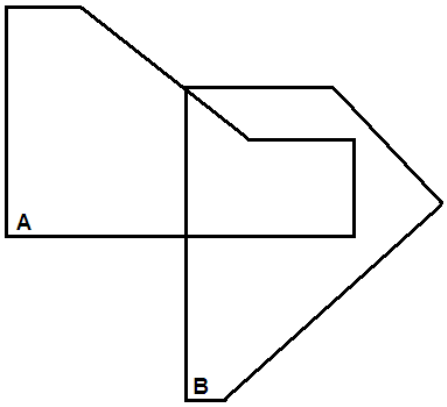

Figura 3.1. Conjuntos A y B

Sea A un conjunto en Z^2. Si a = (a_1, a_2) es un elemento de A, entonces:

$$a \in A$$

Si a no es un elemento de A

$$a \notin A$$

El conjunto sin elementos es llamado conjunto nulo o vacío y se denota por Ø.

Un conjunto se especifica por el contenido de dos llaves { }.

Si cada elemento de un conjunto A es también un elemento de otro conjunto B, entonces se dice que A es un subconjunto de B:

$$A \subseteq B$$

La unión de dos conjuntos A y B se denota:

$$C = A \cup B$$

Es el conjunto de todos los elementos que pertenecen a cualquiera de A, B o a ambos. Similarmente la intersección de dos conjuntos A y B, se denota por:

$$D = A \cap B$$

Es el conjunto de todos los elementos que pertenecen a ambos A y B.

Dos conjuntos A y B se dice que son mutuamente exclusivos si no tienen elementos en común, en este caso:

$$A \cap B = 0$$

El complemento del conjunto A es el conjunto de elementos que no están contenidos en A:

$$A^c = \{w \mid w \notin A\}$$

La diferencia de dos conjuntos A y B, denotada por A-B se define como:

$$A - B = \{w \mid w \in A, w \notin B\} = A \cap B^c$$

La reflexión del conjunto B se denota por $\hat{B}$ y se define como:

La traslación del conjunto A por un punto $z=(z_1,z_2)$ denotada $(A)_z$, se define como:

$$(A)_z = \{c \mid c = a + z, \forall a \in A\}$$

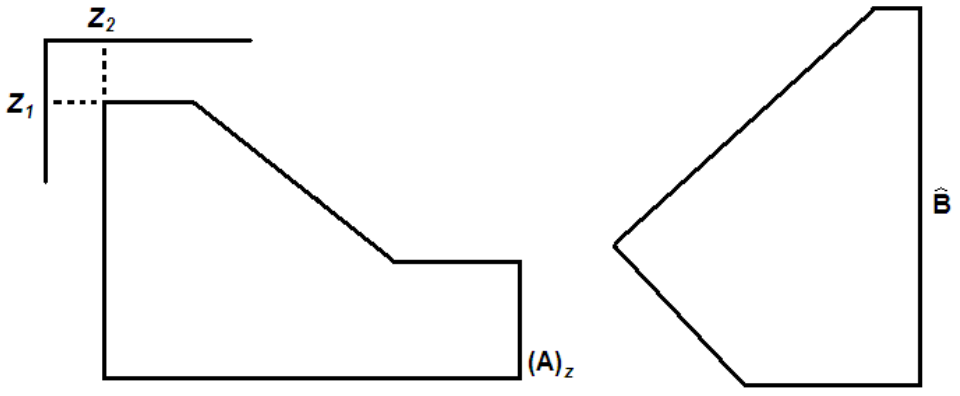

Figura 3.2. Conjunto A trasladado por Z y reflexión del conjunto B

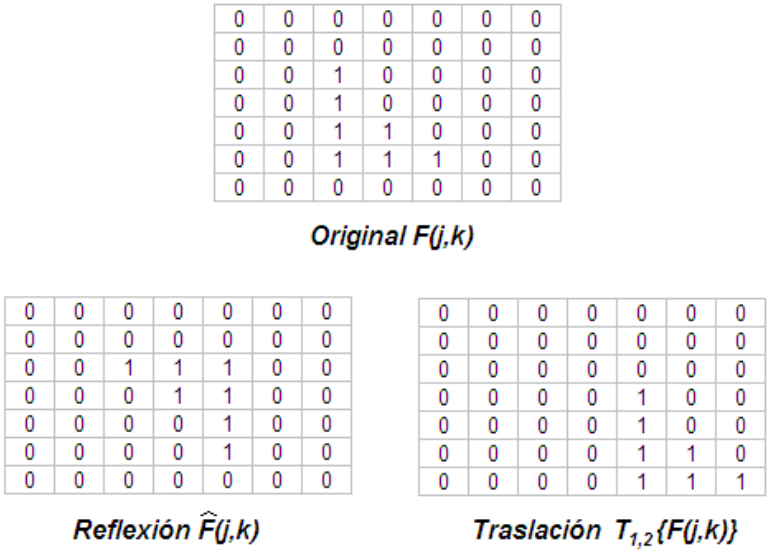

Figura 3.3. Traslación y reflexión en imágenes binarias.

3.01.2 Operaciones lógicas en imágenes binarias.

Las operaciones lógicas proveen un complemento poderoso para la implementación de algoritmos para el procesado de imágenes basados en morfología. Las principales operaciones, AND, OR y NOT se pueden combinar para formar cualquier otra operación lógica. Se realizan píxel por píxel, entre los píxeles correspondientes de dos o más imágenes (excepto NOT, que opera en los píxeles de una imagen única).

p	q	p AND q	p OR q	NOT (p)
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Tabla 3-1. Operaciones lógicas.

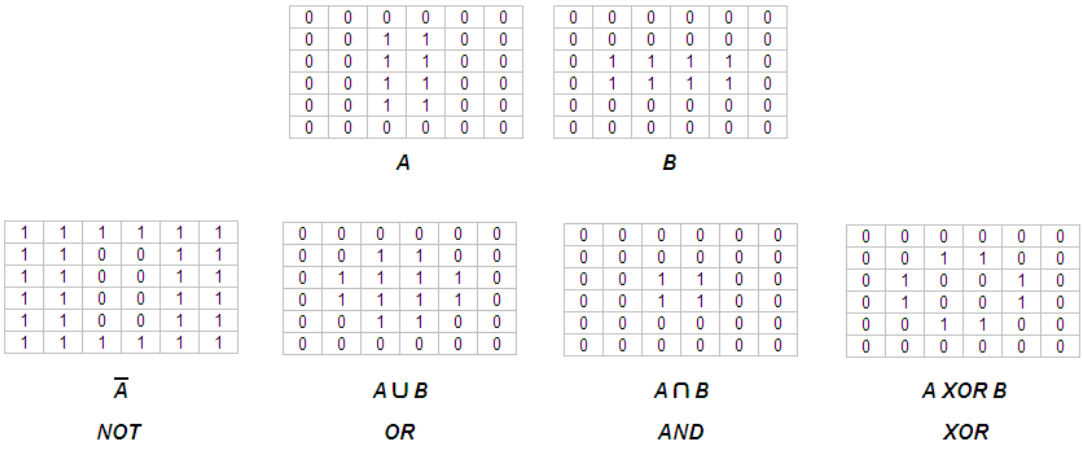

Figura 3.4. Operaciones lógicas en imágenes binarias.

Las operaciones lógicas que se muestran en la tabla 3-1 tienen una correspondencia uno-a-uno con las operaciones de conjuntos, con la limitación de que las operaciones lógicas se limitan a variables binarias, que no es el caso en general para las operaciones de conjuntos. La operación de intersección en la teoría de conjuntos se reduce a la operación AND cuando las variables involucradas son binarias, la operación de unión es la operación OR y la de complemento es la NOT.

La morfología matemática de una imagen binaria se basa en dos operadores, erosión y dilatación.

3.01.3 Dilatación.

Con A y B conjuntos en Z^2, la dilatación de A por B, denotada por:

$$G(j,k) = A(j,k) \oplus B(j,k)$$

$$G = A \oplus B$$

Donde A(j,k) para $1 \leq j,k \leq N$ es una imagen binaria y B(j,k) para $1 \leq j,k \leq L$, donde L es un entero impar. A y B se asumen como arreglos cuadrados.

La dilatación puede ser definida matemáticamente e implementada de varias maneras.

Definición en sumas de Minkowski:

$$G(j,k) = \cup\cup T_{r,c}\{A(j,k)\}$$

$$(r,c) \in B$$

Se dice que G está formada por la unión de todas las traslaciones de A con respecto a sí misma en la que la distancia de la traslación es el índice de fila y columna de B que es 1 lógico. La figura 3.5 muestra el concepto.

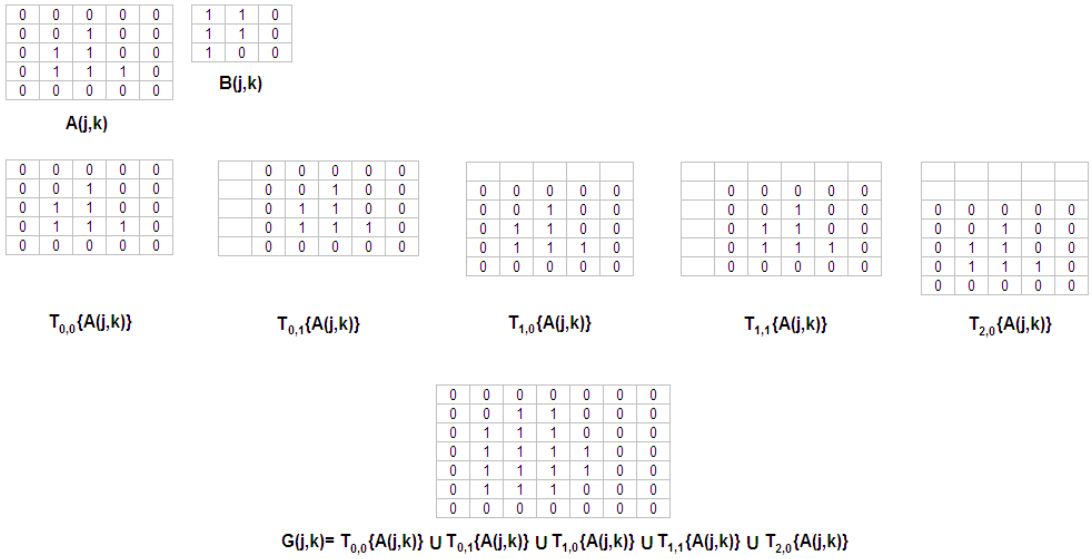

Figura 3.5. Definición de dilatación en sumas de Minkowski.

El arreglo de salida es de dimensión M=N+L-1, donde L es el tamaño del elemento estructurante. Con el fin de registrar las imágenes de entrada y salida correctamente A(j, k) debe ser trasladada diagonalmente a la derecha por Q=(L-1)/2 pixeles.

Otra definición se basa en la obtención de la reflexión de B sobre su origen y el desplazamiento de esta reflexión por z. La dilatación de A por B,

es el conjunto de todos los desplazamientos, z, de manera que A y B coinciden por lo menos en un elemento.

$$A \oplus B = \left\{ z \mid \left(\hat{B}\right)_z \cap A \neq 0 \right\}$$

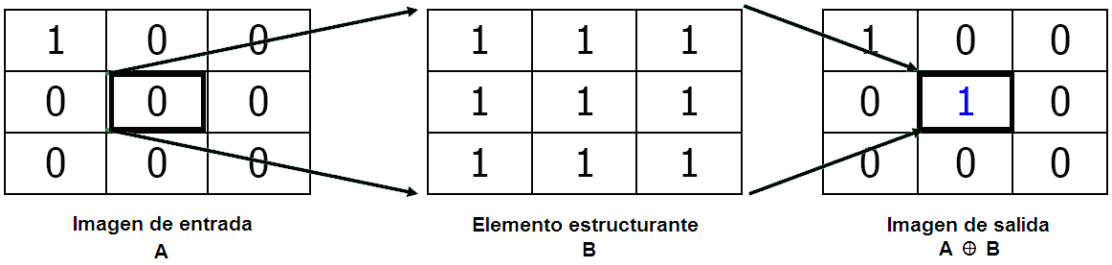

Figura 3.6. Dilatación de A por B.

El conjunto B se llama comúnmente elemento estructurante.

Es interesante comparar la dilatación con la convolución. En la dilatación, la operación de unión es análoga a la operación de suma de la convolución, mientras que la operación de intersección es análoga a la multiplicación punto a punto, como en la convolución, La dilatación puede ser concebida como el escaneo y procesamiento de A por B girado 180°.

Una de las aplicaciones más simples de la dilatación es para llenar huecos. Una ventaja inmediata de la morfología matemática sobre el método del filtro pasa bajas para llenar huecos es que el método morfológico resulta directamente en una imagen binaria. Un filtrado pasa bajos, comienza con una imagen binaria y produce una imagen en escala de grises la cual requiere ser pasada por una función de umbralización para convertirla nuevamente en imagen binaria.

3.01.4 Erosión.

Se expresa simbólicamente:

$$G(j,k) = A(j,k) \Theta B(j,k)$$

$$G = A \Theta B$$

Donde B(j,k) es un elemento estructurante de tamaño impar LxL.

Se define como:

$$G(j,k) = \bigcap\bigcap T_{r,c}\{A(j,k)\}$$

$$(r,c) \in B$$

El significado de esta relación es que la erosión de A por B es la intersección de todas las traslaciones de A en las cuales la distancia de traslación son los índices de la fila y la columna del píxel de B que están en el estado lógico 1.

Otra definición de la erosión de A por B es el conjunto de todos los puntos z tales que B trasladado por z, están contenidos en A.

Para conjuntos A y B en Z^2, se define como:

$$A \Theta B = \{z \mid (B)_z \subseteq A\}$$

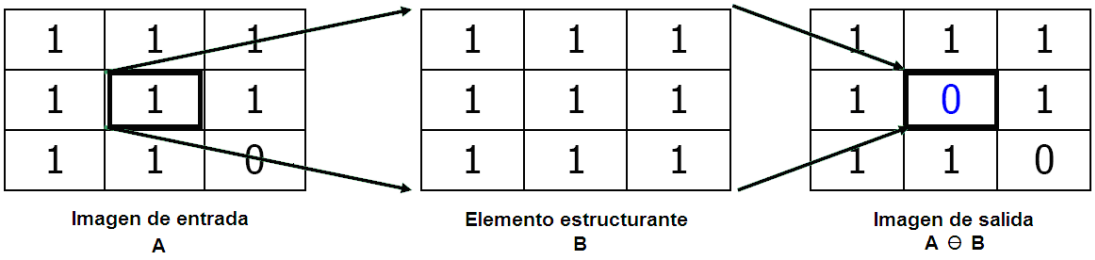

Figura 3.7. Erosión de A por B.

Uno de los usos más simples de la erosión es para eliminar detalles irrelevantes (en términos de tamaño) de una imagen binaria.

Las dos definiciones que se presentan para erosión y las dos de dilatación son equivalentes y se obtienen los mismos resultados.

Mientras que la erosión es un operador que encoge la imagen, la dilatación es un operador de expansión. La erosión y la dilatación no son

invertibles, en general, la dilatación no restaura completamente un objeto erosionado.

La dilatación y la erosión son duales entre sí con respecto a la complementación y reflexión de conjuntos, es decir:

$$(A \Theta B)^c = A^c \oplus \hat{B}$$

3.01.5 Apertura.

La apertura de un conjunto A por un elemento estructurante B, es la erosión de A por B, seguida de la dilatación del resultado por B. se define como:

$$A \circ B = (A \Theta B) \oplus B$$

La operación de apertura tiene una interpretación geométrica simple, supongamos que vemos el elemento estructurante B como una “bola rodante” plana. La frontera de $A \circ B$ se establece por los puntos en B que llegan más lejos en A mientras B rueda en interior de A.

Esta propiedad de “encajar” del operador de apertura conlleva a otra formulación de la apertura, la cual establece que la apertura de A por B se obtiene tomando la unión de todas las traslaciones de B que caben en A. La apertura puede ser expresada como:

$$A \circ B = \cup \{ (B)_z \mid (B)_z \subseteq A \}$$

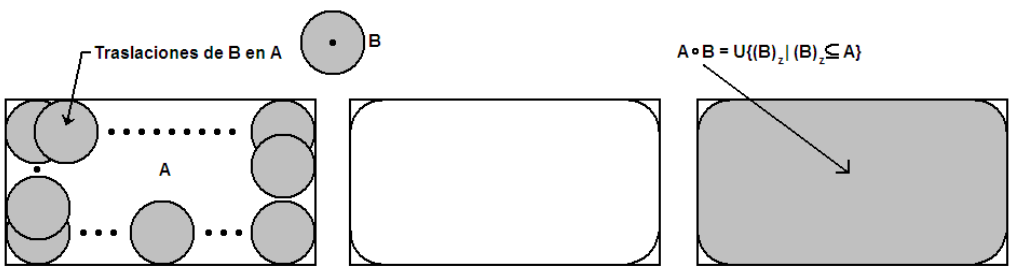

Figura 3.8. Apertura de A por B. B rueda en el interior de A.

Donde $\cup\{\bullet\}$ denota la unión de todos los conjuntos dentro de las llaves.

3.01.6 Cerradura.

La cerradura del conjunto A por un elemento estructurante B, se define como la dilatación de A por B seguida de la erosión del resultado por B, se define como:

$$A \bullet B = (A \oplus B) \Theta B$$

La cerradura tiene una interpretación geométrica similar a la dilatación, excepto que ahora B rueda en el exterior de la frontera. Geométricamente, un punto w es un elemento de $A \bullet B$ si y solo si $(B)_z \cap A \neq 0$ para cualquier traslación de $(B)_z$ que contiene w.

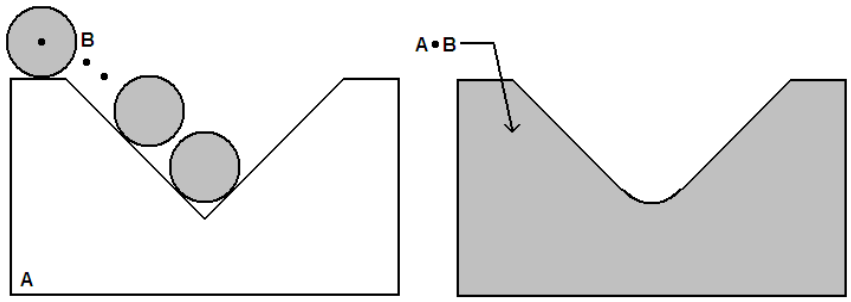

Figura 3.9. Cerradura de A por B. B rueda en el exterior de A.

Propiedades del operador de apertura.

- $A \circ B$ es un subconjunto (subimagen) de A
- Si C es un subconjunto de D, entonces $C \circ B$ es un subconjunto de $D \circ B$
- $(A \circ B) \circ B = A \circ B$

Propiedades del operador de cerradura.

- A es un subconjunto (subimagen) de $A \bullet B$

- Si C es un subconjunto de D, entonces $C \bullet B$ es un subconjunto de $D \bullet B$
- $(A \bullet B) \bullet B = A \bullet B$

La apertura generalmente suaviza el contorno de un objeto, rompe istmos estrechos, y elimina salientes delgadas. La cerradura también tiende a suavizar secciones de contorno, pero, a diferencia de la apertura, por lo general, une aberturas estrechas, elimina los agujeros pequeños, y llena lagunas en el contorno.

En el operador de apertura la dilatación trata de deshacer la operación de erosión, sin embargo algunos detalles estrechamente relacionados a la forma y el tamaño del elemento estructurante desaparecen. Un objeto que desaparece por consecuencia de una erosión no puede ser recuperado.

Las operaciones morfológicas pueden ser usadas para construir filtros similares en concepto a filtros espaciales.

3.01.7 Granulometría.

Granulometría es un campo que se ocupa principalmente de la determinación de la distribución de tamaño de partículas en una imagen. La figura 3.10 muestra una imagen compuesta de objetos iluminados de tres tamaños diferentes. Los objetos no sólo se solapan, sino que también están demasiado desordenados para permitir la detección de las partículas individuales. Debido a que las partículas son más iluminadas que el fondo, el enfoque morfológico siguiente se puede utilizar para determinar la distribución de tamaño. Operaciones de apertura con un elemento estructurante de tamaño creciente se realizan en la imagen original. La diferencia entre la imagen original y su apertura se calcula después de cada pasada. Al final del proceso, estas diferencias están normalizadas y se utilizan para construir un histograma de la distribución de tamaño de las

partículas. Este enfoque se basa en la idea de que las operaciones de apertura de un tamaño determinado tienen el mayor efecto en las regiones de la imagen que contienen partículas de tamaño similar. Por lo tanto, una medida del número relativo de tales partículas, se obtiene calculando la diferencia entre las imágenes de entrada y salida.

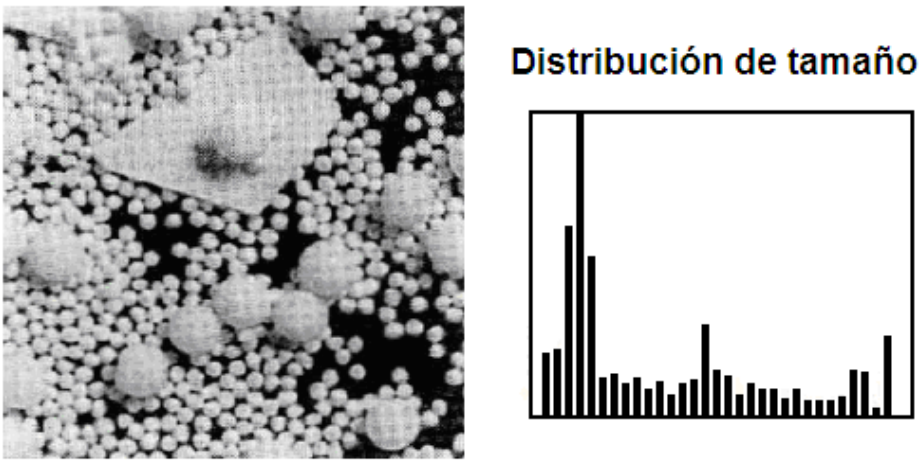

Figura 3.10. Distribución de tamaño en una imagen con objetos de diferentes tamaños.

Medir el cambio en el área de un objeto binario, cuando sucesivas aperturas son aplicadas, nos provee de un medio para evaluar los componentes morfológicos descriptores de la forma del objeto, llamado espectro de patrones o pecstrum. En un espectro de patrones la desaparición progresiva de la imagen es numéricamente capturada midiendo la diferencia en el área entre cada paso. El pecstrum descompone la imagen en componentes morfológicos de acuerdo a la forma y al tamaño del elemento estructurante, provee un análisis cuantitativo del contenido morfológico de la imagen.

La forma discreta del espectro de patrones esta dado por:

$$P(n,B)=\frac{M[A\circ nB]-M[A\circ(n+1)B]}{M[A]}$$

Donde M representa el área medida en operaciones intermedias, y nB es el elemento estructurante dilatado n veces.

El pesctrum tiene la propiedad de ser invariante a la rotación y translación cuando B es un elemento estructurante isotrópico. La escala es determinada por el tamaño del elemento estructurante.

La figura 3.11 presenta el método en una forma grafica. El espectro de patrones presenta inestabilidad cuando ruido o una cuadricula rectangular discontinua afectan a la imagen binaria. En este caso el valor de algunos componentes espectrales cambia significativamente.

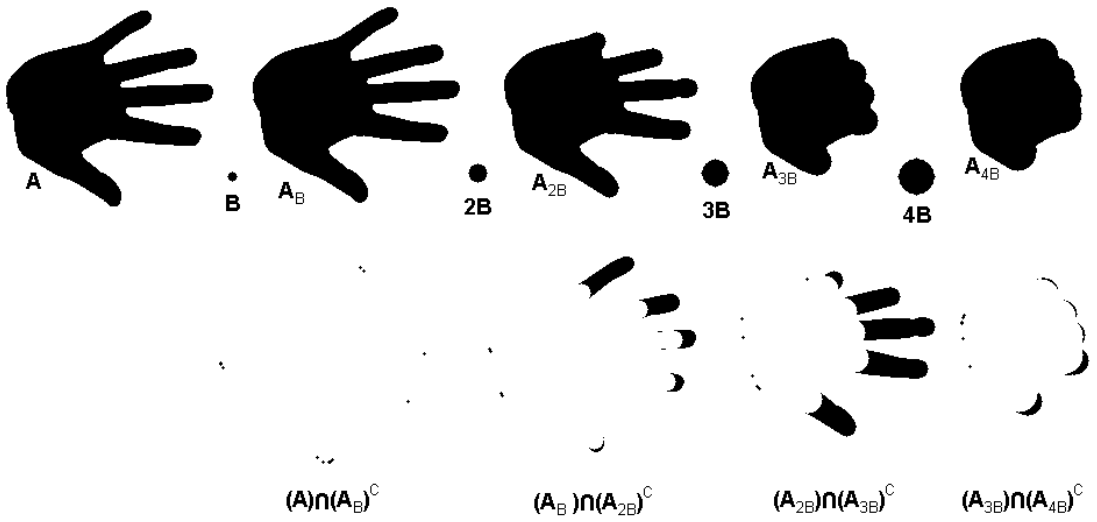

Figura 3.11. Pecstrum. Aperturas sucesivas con el elemento estructurante creciente.

Capitulo 4.

Diseño

4.01 Descripción general del programa.

El programa se implementó en una tarjeta nexy2 de digilent, basada en un fpga Xilinx Spartan 3E. Con una frecuencia de trabajo de 50 MHz. El procesamiento realizado consistirá en la aplicación sucesiva del filtro de apertura con un kernel de mayor radio entre aperturas siendo el radio máximo del kernel de 32 (65x65)

Aprovechando el diseño modular de VHDL, el sistema de procesamiento se dividió en bloques más pequeños: comunicación con la PC, memoria para guardar la imagen original y procesada, procesamiento de la imagen y un bloque para la resta de las áreas y el envío del vector de resultados a la PC. Los bloques de comunicación y de procesamiento de la imagen hacen uso de la memoria, se diseñaron bloques de multiplexores que ceden el control de la memoria a cada bloque según lo requiera el programa.

4.01.1 Bloques para la comunicación con la PC.

Se diseñaron los bloques necesarios para la comunicación del FPGA con la PC, guardar la imagen en la memoria RAM del FPGA y para enviar la imagen procesada a la PC usando el puerto serial de la tarjeta. Para recibir la imagen original y enviar la imagen procesada se requiere de los siguientes bloques:

- Una unidad USART (universal synchronous asynchronous receiver transmitter) para enviar y recibir datos de forma serial.
- Un reloj que genere el baud rate necesario para la transmisión y recepción de datos.
- Un bloque que recibe la imagen original y envía la imagen procesada a la computadora.

4.01.2 Bloque de memoria.

La imagen por ser una imagen binaria ocupa menos espacio que una imagen en escala de grises y se puede se guardar en la memoria RAM interna de FPGA. El tamaño de la memoria es el doble del tamaño de la imagen. En la parte baja se guarda la imagen original, así como, el resultado de la dilatación (el resultado de la apertura) y en la parte alta se guarda el resultado de la erosión. Bloques necesarios:

- Memoria RAM para almacenar la imagen original y la imagen procesada.

4.01.3 Bloques de procesamiento.

Como el acceso a los datos de la memoria RAM se hace de uno en uno para la lectura o escritura se decidió realizar de la misma forma el procesado de la imagen, en lugar de hacer las operaciones con las matrices completas el procesamiento de la imagen se realiza elemento (submatriz de la imagen) a elemento (kernel), esto es, conforme se van obteniendo los datos de la memoria se van procesando con los datos del kernel. Todos los bloques que forman el bloque de procesamiento están diseñados para trabajar elemento a elemento de la submatriz y del kernel.

Del capitulo 1 morfología matemática, las operaciones morfológicas necesitan dos operandos, un kernel o ventana que es básicamente una matriz que se utiliza para verificar correspondencia y una submatriz de pixeles de la imagen que tiene como punto central al píxel (A, B) que se esta procesando.

Un bloque genera la posición (F, C) de cada elemento de una submatriz del tamaño del kernel, desde (0,0) hasta (kernel-1, kernel-1), donde “kernel” es un numero impar y define el tamaño del kernel.

El bloque kernel utiliza las coordenadas (F, C) de la submatriz para generar los valores correspondientes al kernel.

Un bloque genera las coordenadas de los pixeles que se procesan (A_p, B_p), estas mismas se utilizan como base para generar la dirección donde se va a guardar el píxel procesado.

Ya que la imagen es en sí misma una matriz de datos y las memorias son lineales, es necesario hacer un bloque que en base a las coordenadas (A, B) del píxel entregue la dirección donde se encuentra guardado en la memoria RAM.

Otro bloque se encarga de hacer un “mapeo” de los índices de la submatriz (F, C) con los correspondientes a la imagen (A, B), utilizando como píxel central al píxel que se procesa (A_p, B_p).

El bloque de operaciones las realiza en forma elemento (kernel)-elemento (submatriz) para cada píxel de la imagen original. Puede realizar la operación de erosión o dilatación según se requiera, lo que permite implementar la operación de apertura, así como la de cerradura. También lleva la cuenta del área de la imagen, esto es, tiene un contador que se incrementa cada vez que el resultado de la apertura de un píxel da como resultado un 1.

Finalmente, es necesario un bloque que controle que el procesamiento se realice para cada píxel de la imagen y que informe cuando se ha finalizado el procesamiento de la imagen original

Por lo tanto, para procesar un píxel se requieren los siguientes bloques:

- Bloque kernel.
- Bloque generador de direcciones del píxel a procesar.
- Bloque generador de índices de la submatriz.

- Bloque que “mapea” los índices de la submatriz (F, C) a los de la imagen (A, B).
- Bloque que indique la posición de un píxel en la memoria utilizando las coordenadas (A, B) de la imagen.
- Bloque de operaciones.
- Bloque de control de todo el proceso.

4.01.4 Bloque de áreas y envío de vector de resultados.

El bloque de procesamiento entrega el área de la imagen que resulta después de cada apertura. Se necesita de un bloque que haga las restas de las áreas entre aperturas, de esta manera, cuando se termina el procesamiento de la imagen, todos los resultados obtenidos forman el vector de resultados. Otro bloque se encarga de enviar a la PC el vector de resultados. Se necesitan los siguientes bloques

- Bloque de resta de áreas.
- Bloque para enviar el vector de resultados a la PC.

4.02 Comunicación con la PC.

Se hizo un bloque USART que utiliza el puerto serial de la tarjeta nexys-2 para comunicación con la PC. El baud rate se configuró a 9600.

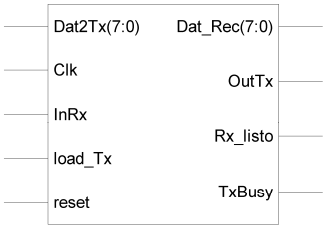

Figura 4.1. Bloque de comunicación.

4.02.1 Generador de baud rate y USART.

El bloque baud_rate genera un reloj que es 16 veces más rápido que la velocidad de transmisión deseada (clk_16). Si se quiere transmitir a 9600, este reloj debe funcionar a 9600*16=153600. El reloj del transmisor (clk_tx) utiliza este reloj para generar las señales de transmisión a 9600 cuenta 16 pulsos del reloj clk_16 y genera un pulso (153600/16). El reloj del receptor (clk_rx) lo utiliza para generar un reloj 2 veces más rápido que la velocidad deseada (153600/8). El receptor una vez que se recibe el bit de inicio resetea el reloj (clk_rx), se cuenta un pulso y al siguiente se muestrea la entrada, se cuenta otro pulso y se vuelve a muestrear la entrada, así hasta recibir todos los bits. Este reloj debe ser 4x la velocidad del baud rate.

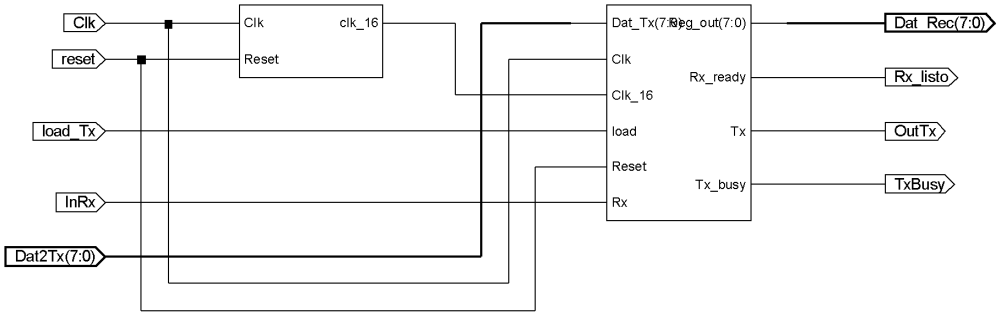

Figura 4.2. Interior del bloque de comunicación, bloques USART y reloj.

4.02.2 Envío y recepción de la imagen.

El bloque load_send1 genera las direcciones, configura las señales de enable, re y we de la memoria para guardar o leer los datos. Guarda los datos que recibe el usart en la memoria; lee la imagen procesada y genera la señal para cargar el dato en el USART para enviarlo a la PC.

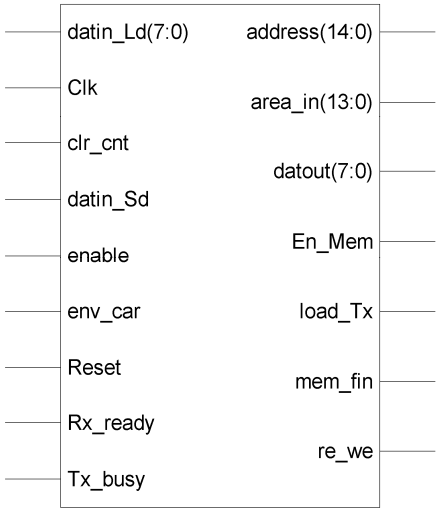

Figura 4.3. Bloque de envío y recepción de la imagen.

datin_Ld – dato que se guarda en la memoria RAM.

datin_Sd – dato que se envía a la PC.

env_car – indica si se envía o se recibe.

Rx_ready – indica que se ha recibido un dato.

Tx_busy – indica si el bloque USART esta enviando un dato.

address – bus de direcciones para la memoria.

area_in – área de la imagen original.

datout – dato que se envía por el USART.

En_Mem – señal para habilitar la memoria

re_we – habilita lectura o escritura en la memoria RAM.

mem_fin – indica que se recibieron o enviaron todos los datos.

load_tx – señal para cargar datout en el bloque del usart.

4.03 Memoria.

Se diseño el bloque de memoria para que el FPGA utilice la memoria RAM que trae implementada. En una sola memoria se guarda la imagen original y la procesada, por lo que el tamaño de la memoria debe ser del doble del tamaño de la imagen. En la parte baja se guarda la imagen original, así como, el resultado de la dilatación (el resultado de la apertura) y en la parte alta se guarda el resultado de la erosión.

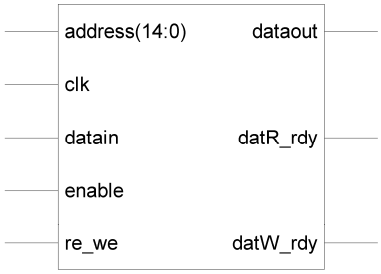

Figura 4.4. Bloque de memoria.

address – direcciones de la memoria.
datin – dato a guardar en la memoria
re_we – lectura o escritura.
dataout – dato leído de la memoria.
datR_rdy – indica que el dato esta listo en la salida (dataout).
datW_rdy – indica que el dato ya se guardo.

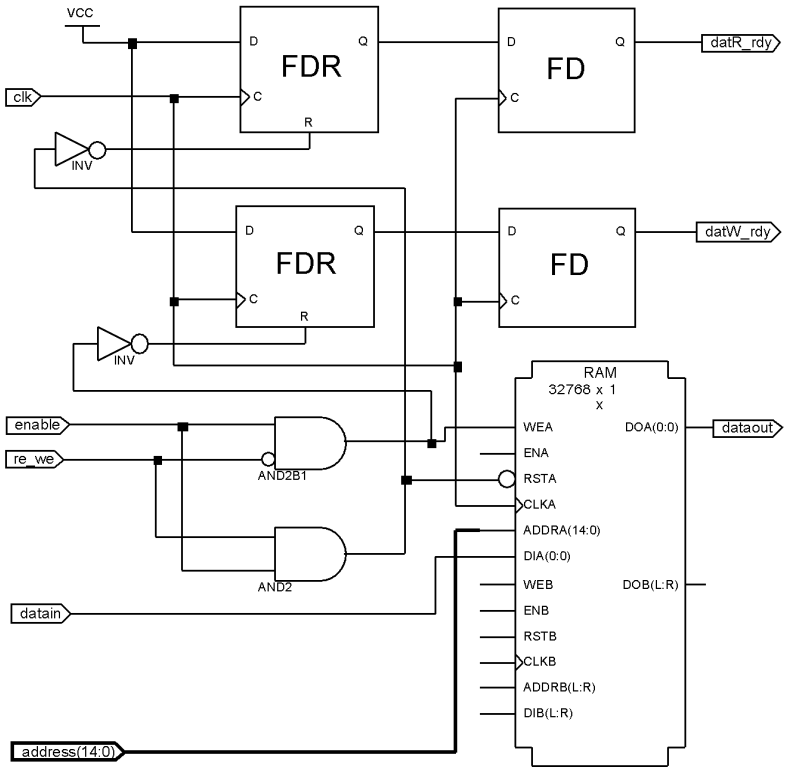

Figura 4.5. Interior del bloque de memoria, se muestra la memoria del FPGA que se utiliza.

4.04 Mux's para el direccionamiento de la memoria.

Los multiplexores se utilizan para cambiar las señales de control (enable, RE, WE), de datos y de direcciones (ADDR) de la memoria entre los bloques de procesamiento y comunicación.

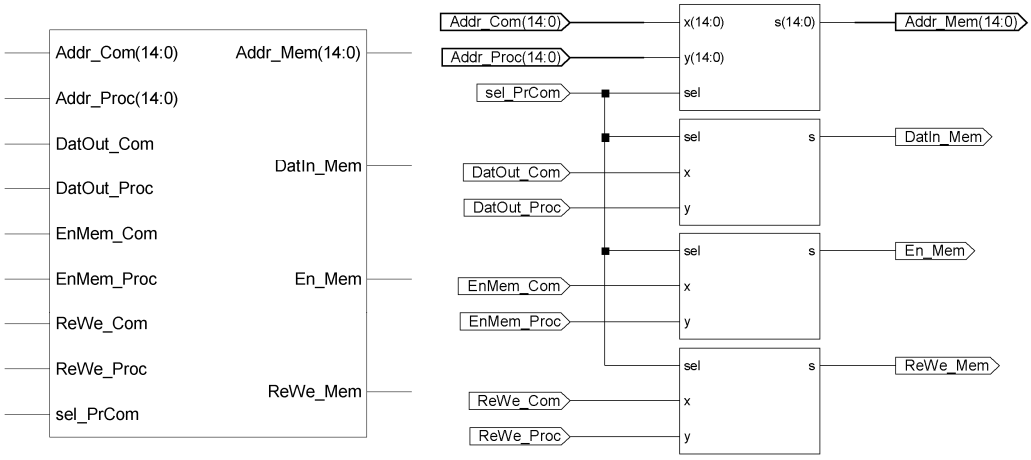

Figura 4.6. Bloque de multiplexores.

4.05 Bloque ctrl3b.

El bloque ctrl3b se utiliza para seleccionar los diferentes modos de funcionamiento del programa:

- Recibe una imagen.
- Inicia procesamiento de la imagen
- Envía la imagen procesada.
- Envía el vector de resultados.

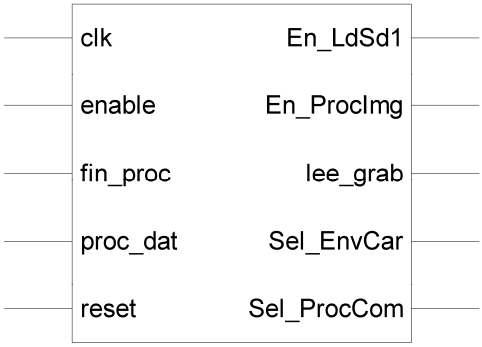

Figura 4.7. Bloque ctrl3b.

4.06 Procesamiento de la imagen.

La imagen a procesar se guarda en la parte baja de la memoria RAM. El bloque de procesamiento realiza una operación de erosión que comienza con un kernel de 3x3, el resultado de la erosión se guarda en la parte alta de la RAM, posteriormente se realiza una operación de dilatación, con el mismo kernel y el resultado se guarda en la parte baja de la memoria, de esta manera, en la parte baja de la RAM se tiene el resultado de una operación de apertura, el tamaño del kernel se incrementa y se repiten las operaciones.

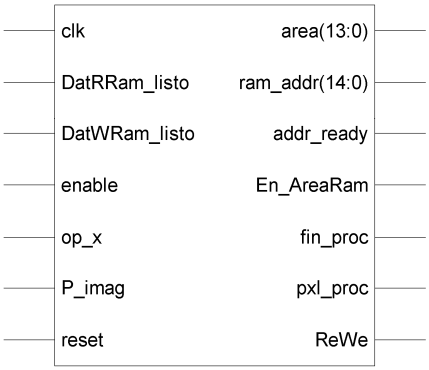

Figura 4.8. Bloque de procesamiento.

DatRRam_listo – indica que ya esta listo un dato leído de la memoria.

DatWRam_listo - indica que ya se grabó el dato en la memoria.

op_x – dato de la memoria, representa un píxel de la imagen.

P_imag – señal de inicio para el procesado de la imagen.

area – es el área de la imagen después de cada apertura.

ram_addr – bus de direcciones de la memoria.

addr_ready – indica que hay una dirección valida en el bus de direcciones.

En_AreaRam – habilita memoria.

ReWe – lectura o escritura en la memoria.

pxl_proc – píxel procesado.

fin_proc – indica que ya se proceso la imagen.

4.06.1 Bloques que forman al bloque de procesamiento.

4.06.1.1 Bloque LeeGrab_AddrRam2.

El bloque LeeGrab_AddrRam2 genera las direcciones de la memoria RAM a partir de los índices (A, B) que indican la posición del píxel en la matriz de la imagen.

La imagen se guarda en la memoria RAM en forma lineal, por lo que en lugar de tener un arreglo matricial se tiene un vector, la tabla 4-1 muestra

la distribución de los pixeles de una imagen de 128x128, los números que se encuentran debajo de los índices del píxel indican la posición en la memoria RAM. Los pixeles del (0,0) al (0,127) se guardan en las direcciones 0 al 127 de la RAM y así sucesivamente. La dirección del píxel en la memoria se obtiene de multiplicar el número de columnas de la imagen por A y al resultado se le suma B.

$$addr = NxA + B$$

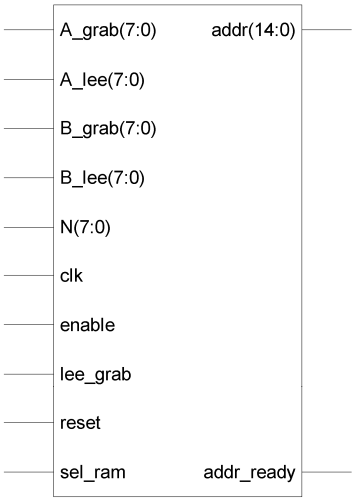

Figura 4.9. Bloque LeeGrab_AddrRam2.

(A,B) (0,0) 0	(0,1) 1	(0,2) 2	. . .	(0,127) 127
(1,0) 128	(1,1) 129	(1,2) 130	. . .	(1,127) 255
(2,0) 256	(2,1) 257	(2,2) 258		(2,127) 383
(127,0) 16256	(127,1) 16257	(127,2) 16258		(127,127) 16383

Tabla 4-1 Distribución de pixeles en la memoria.

A_grab, B_grab - vienen del bloque cont_pxl y se utilizan para generar la dirección donde se guarda el píxel procesado en la memoria RAM.

A_Lee, B_Lee - vienen del bloque sub_mat y se utilizan para generar las direcciones que se leen de la memoria RAM.

lee_grab - indica cual par (A, B) se utiliza para generar la dirección de la RAM.

sel_ram selecciona si se utiliza la parte baja o la alta de la memoria.

N – número de columnas de la imagen.

addr – dirección del píxel en la memoria.

addr_ready – indica que ya esta lista la dirección en la salida.

4.06.1.2 Bloque cont_pixel.

El bloque cont_pixel se encarga de generar las coordenadas del píxel a procesar (A_p, B_p), va de (0,0) hasta (n-1, m-1). Cada vez que la señal Next_pxl va a 1 se pasa al siguiente píxel, cuando se alcanza el último la señal de last_pxl va a 1 para indicar que ya se proceso toda la imagen. Enable se utiliza para habilitar el bloque, si enable se hace 0, los contadores se reinician, (A, B) =(0,0), para procesar de nuevo la imagen.

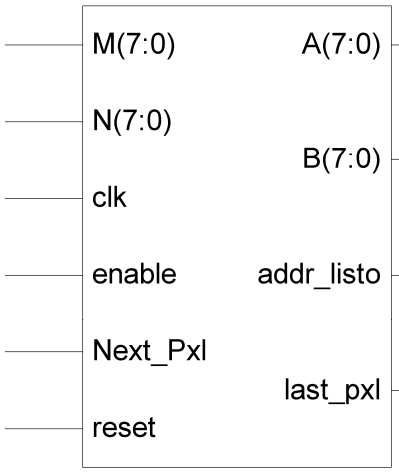

Figura 4.10. Bloque cont_pxl.

(A,B) (0,0)	(0,1)	(0,2)	. . .	(0,127)
(1,0)	(1,1)	(1,2)	. . .	(1,127)
(2,0)	(2,1)	(2,2)		(2,127)
(127,0)	(127,1)	(127,2)		(127,127)

Tabla 4-2 Distribución de pixeles en la imagen.

M - número de filas de la imagen

N - número de columnas de imagen

A, B - índices del píxel de la imagen

addr_listo – se utiliza para indicar que los índices A y B ya están en la salida.

last_pxl – indica que los índices A y B corresponden al último píxel de la imagen.

Next_pxl – se utiliza para avanzar los pixeles.

enable – se utiliza para habilitar el bloque, si enable se hace 0, los contadores se reinician, (A, B) =(0,0).

4.06.1.3 Bloque ctrl_submat1.

El bloque ctrl_SubMat genera los índices de una submatriz de tamaño k (2*radio+1) cuando la señal sig_k va a 1 se pasa al siguiente elemento de la fila, va desde (0,0) hasta (k-1, k-1) cuando está en el último elemento de la submatriz la señal sub_matF va a 1 para indicar que ya se procesó toda la submatriz. Cuando enable se hace 0 los registros se borran para comenzar con una nueva submatriz.

Esta submatriz se utiliza para generar los valores de la imagen que rodean al píxel que se procesa, y para obtener los valores del kernel.

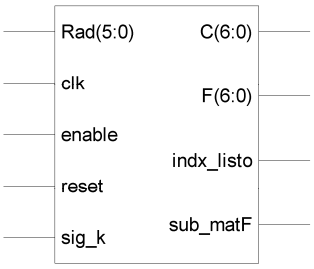

Figura 4.11. Bloque ctrl_submat1.

(F,C) (0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)

Tabla 4-3 Submatriz de 3x3.

K – indica el tamaño del kernel.

C, F - índices de la submatriz (también se utilizan para obtener los valores del kernel).

Indx_listo – indica que los índices ya están en la salida.

sub_matF – indica que los índices (F, C) corresponden al último elemento del kernel.

sig_k - se utiliza para incrementar los índices.

enable – enable se utiliza para habilitar el bloque y para borrar los contadores (F, C) = (0,0) para generar otra submatriz.

4.06.1.4 Bloque Kernel_var

El bloque kernel_var saca los valores del Kernel de acuerdo a las coordenadas (F, C) y al tamaño del Kernel (definido por K_radio). La forma del kernel es un círculo de radio K_radio.

Para obtener los valores del kernel el bloque implementa la ecuación:

$$k\_dato = \begin{cases} \left(C^2 + F^2\right) \le k\_radio^2 \Rightarrow k\_dato = 1 \\ \left(C^2 + F^2\right) > k\_radio^2 \Rightarrow k\_dato = 0 \end{cases}$$

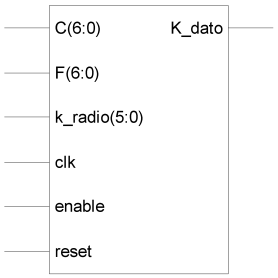

Figura 4.12. Bloque kernel_var.

0	1	0
1	1	1
0	1	0

Tabla 4-4. Kernel.

F, C - índices del kernel.

K – Define el tamaño del kernel

K_dato – valor del kernel en la posición (F, C)

Enable – con enable en alto la salida cambia cuando cambian las entradas.

4.06.1.5 Bloque oper1

El bloque oper1 hace las operaciones de erosión y dilatación. Está programado para una apertura (una erosión seguida de una dilatación). Lleva la cuenta del área de la imagen entre cada operación de apertura.

De la definición de erosión

$$A \Theta B = \left\{ z \mid \left(B\right)_z \subseteq A \right\}$$

Se tiene que la imagen debe coincidir con el kernel en todas las posiciones donde el kernel sea 1, si al menos una posición no coincide, el resultado de ese píxel es 0.

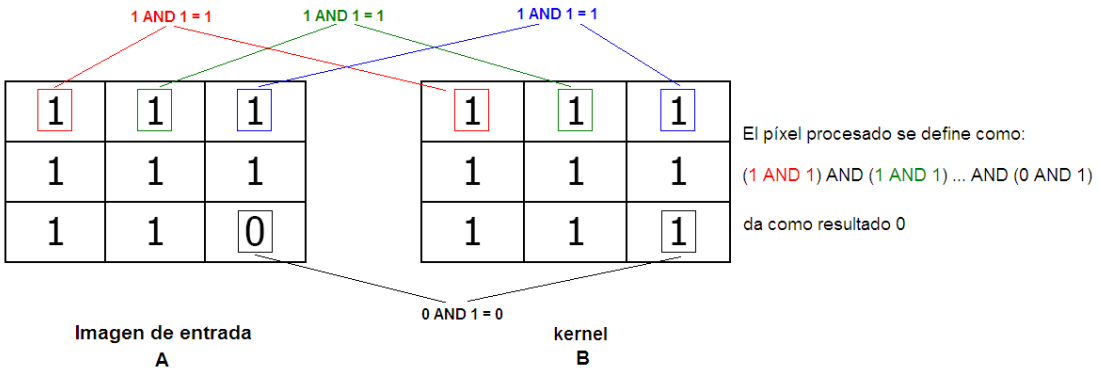

Figura 4.13. Verificando correspondencia para operación de erosión.

Solo se hacen operaciones en donde el kernel sea 1, se hace una and de op_2 (kernel) con op_1 (imagen) y se guarda el resultado para procesarlo con el siguiente producto. En el caso de la erosión al resultado se le hace una AND con el resultado anterior. Si después de procesar toda la submatriz el resultado es 1 equivale a decir que todos los pixeles de la submatriz y del kernel coincidieron.

De la definición de dilatación:

$$A \oplus B = \left\{ z \mid \left(\hat{B}\right)_z \cap A \neq 0 \right\}$$

Se tiene que si la imagen coincide con el kernel en al menos una posición, el resultado del procesamiento de ese píxel es 1.

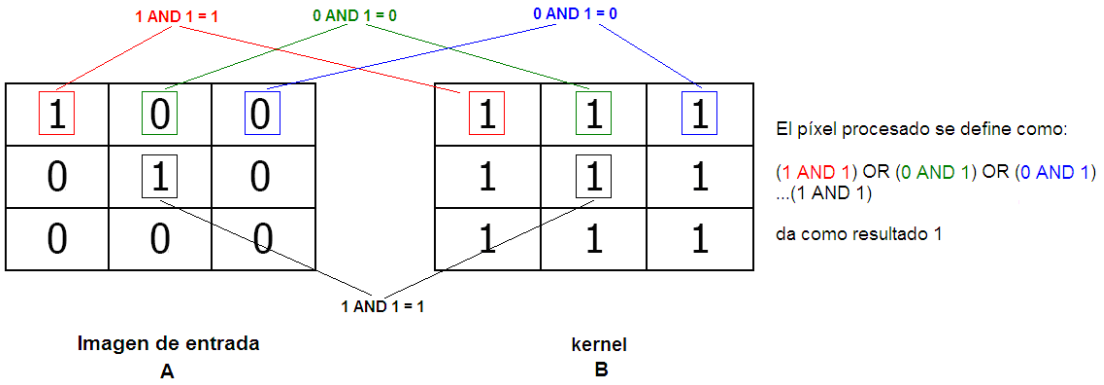

Figura 4.14. Verificando correspondencia para operación de dilatación.

De la misma manera que para la erosión, solo se hacen operaciones en donde el kernel sea 1. Al resultado se le hace una OR con el resultado anterior. Si después de procesar toda la submatriz el resultado es 1 equivale a decir que al menos un píxel de la submatriz y del kernel coincidió.

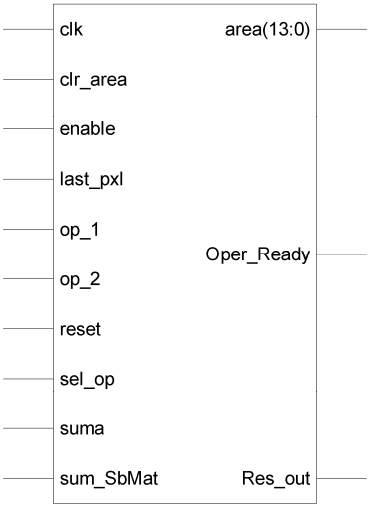

Figura 4.15. Bloque oper1.

clr_area – se utiliza para borrar el contador del área de la imagen.

enable – con enable en 0 se borran los registros de las operaciones. Inicia el procesamiento de otro píxel.

last_pxl – se usa para saber si el píxel que se procesa es el último de la imagen.

op_1 – píxel de la imagen.

op_2 – elemento del kernel.

sel_op – se utiliza para seleccionar que tipo de operación realiza erosión o dilatación.

suma – indica que se tienen nuevos valores en op_1 y op_2 y se realiza la operación correspondiente.

sum_SbMat – indica que ya se proceso una submatriz completa.

area – área de la imagen entre cada operación.

oper_ready – indica que ya se tiene el resultado de una operación.

Res_out – es el resultado de procesar un píxel.

4.06.1.6 Bloque sub_mat

El bloque sub_mat se encarga de asignar la submatriz generada por el bloque ctrl_submat1 a la imagen, tomando como píxel central al (A_p, B_p). No llena la submatriz con los valores de la imagen si no que genera las coordenadas del píxel que corresponden a cada elemento de la submatriz.

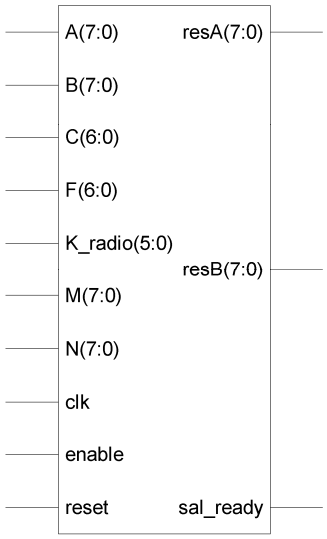

Figura 4.16. Bloque sub_mat.

A, B – índices que definen un píxel de la imagen.

F, C – índices que definen un elemento de la submatriz.

M, N – definen el tamaño de la imagen.

K_radio – radio del kernel, se utiliza para mapear la submatriz.

resA, resB – índices que “mapean” la imagen al elemento (F,C) de la submatriz.

sal_ready – indica que las salidas están listas.

enable – se utiliza para actualizar las salidas. El bloque se ejecuta una sola vez con enable en alto.

En la figura 4.17 se observa la imagen (gris) y la submatriz (gris claro). Cuando se procesa el píxel (2,2) la submatriz “cae” en el área señalada con gris claro.

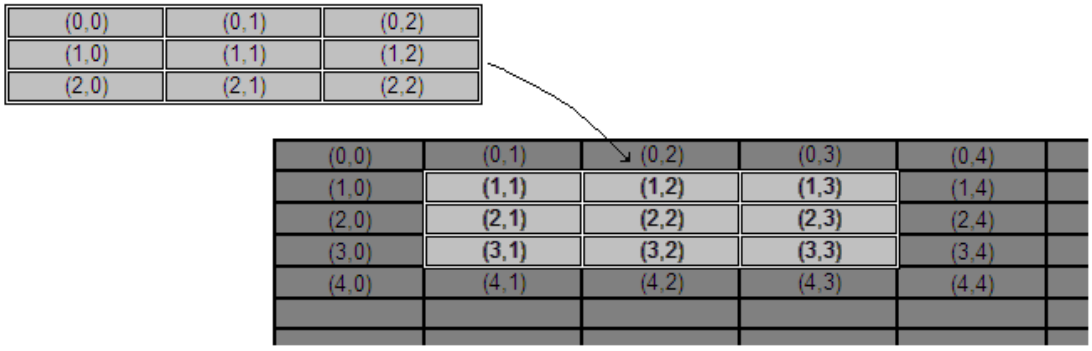

Figura 4.17. Imagen (azul) y submatriz (azul cielo) cuando el píxel procesado es el (2,2).

De la figura 4.17 se obtiene la tabla de correspondencia o de mapeo (tabla 4-5). Al elemento (0,0) de la submatriz le corresponde el píxel (1,1) de la imagen, al (1,1) el (2,2), etc.

(0,0) ⇒ (1,1)	(0,1) ⇒ (1,2)	(0,2) ⇒ (1,3)
(1,0) ⇒ (2,1)	(1,1) ⇒ (2,2)	(1,2) ⇒ (2,3)
(2,0) ⇒ (3,1)	(2,1) ⇒ (3,2)	(2,2) ⇒ (3,3)

Tabla 4-5. Tabla de correspondencia entre la submatriz y la imagen.

Partiendo de la tabla 4-5, se obtienen las siguientes ecuaciones de mapeo:

$$resA:=A_p\text{-}L+F$$
$$res\_B:=B_p\text{-}L+C;$$

donde (A_p ,B_p) definen el píxel a procesar, (F, C) definen al elemento de la submatriz, y L es el radio del kernel.

El bloque sub_mat ya considera los bordes de la imagen, en la figura 4.18 se observa la imagen y las submatrices correspondientes a los pixeles (0,0), (0,127), (127,0) y (127,127).

Para los elementos de la submatriz que quedan fuera de la imagen, por ejemplo el elemento (0,2) cuando el píxel procesado es el (0,127), se copiaron las coordenadas del píxel mas cercano a esa posición, de esta manera, el bloque sub_mat le asigna el píxel (0,127) al elemento (0,2) de la submatriz. Se podría decir que hace la orilla más gruesa.

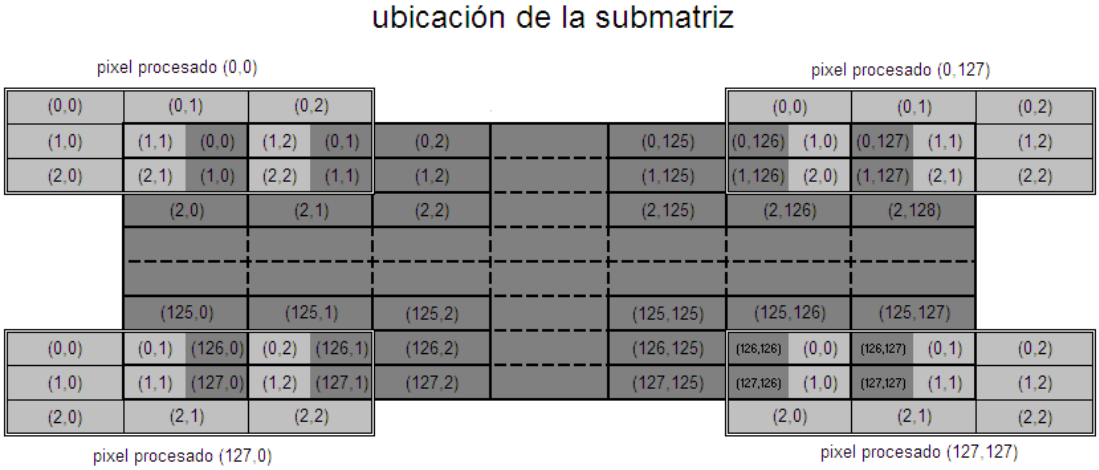

Figura 4.18. La submatriz en las cuatro esquinas de la imagen.

4.06.1.7 Bloque ctrl_proc1.

El bloque ctrl_proc1 se encarga de sincronizar los bloques que hacen el procesado de la imagen, incrementa el tamaño del kernel después de cada apertura, selecciona el tipo de operación que se va a realizar (erosión o dilatación), indica en que parte de la memoria se lee y en que parte se escribe.

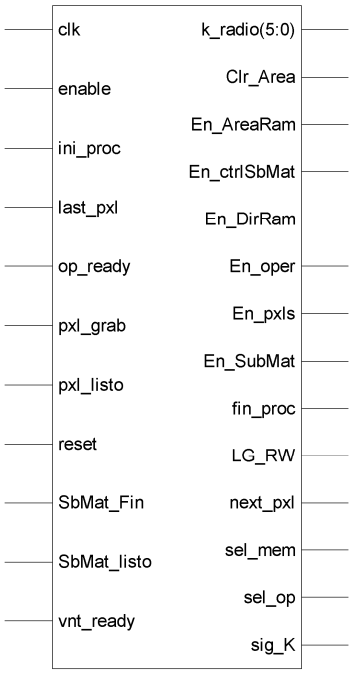

Figura 4.19. Bloque ctrl_proc1.

ini_proc – se usa para iniciar el procesamiento de la imagen,

last_pxl - indica que se esta procesando el ultimo píxel de la imagen.

op_ready - indica que la operación de los elementos ya se completo.

pxl_grab - indica que ya se grabo el píxel procesado.

pxl_listo - indica que las coordenadas del siguiente píxel a procesar ya están listas.

SbMat_Fin - indica que se esta procesando el último elemento de la submatriz.

SbMat_listo - indica que las coordenadas del siguiente elemento de la submatriz ya están listas.

vnt_ready - indica que el “mapeo” ya esta listo.

k_radio - indica el tamaño del kernel.

clr_area - limpia el registro donde se va guardando las áreas entre aperturas.

En_AreaRam - se utiliza para indicar que el área de una apertura esta listo. El bloque area_block1 hace la resta de areas y guarda el resultado en memoria.

En_ctrlSbMat - habilita al bloque ctrl_submat1. Inicia la generación de los elementos de la submatriz.

En_DirRam – habilita el bloque LeeGrab_AddrRam2, genera la dirección de la memoria.

En_oper - habilita al bloque oper1, cuando enable se hace 0 los registros del bloque se hacen 0.

En_SubMat - habilita al bloque sub_mat. Hace el “mapeo”

fin_proc – se utiliza para encender un led. Indica que ya se proceso la imagen.

LG_RW – configura la lectura o escritura de la memoria.

next_pxl – le indica al bloque cont_pxl que genere las coordenadas del siguiente píxel a procesar.

sel_mem – indica cual porción de la memoria se utiliza para leer y cual para grabar.

sel_op – selecciona el tipo de operación que va a realizar el bloque oper1.

sig_k – le indica al bloque ctrl_SubMat1 que genere las coordenadas del siguiente elemento de la submatriz.

4.07 Diagrama simplificado del bloque de procesamiento.

(1) El bloque cont_pxl genera las coordenadas (A_p, B_p) del píxel a procesar, estas coordenadas se utilizan para generar la dirección donde se va a guardar el píxel procesado en la memoria RAM y como píxel central de la submatriz de la imagen.

(2) El bloque ctrl_submat genera las cordenadas (F, C) de los elementos de una submatriz que se utiliza para el procesamiento de cada píxel.

(3) El bloque sub_mat “mapea” las coordenadas (F, C) de la submatriz con las coordenadas (A, B) de los pixeles de la imagen, tomando como píxel central al (A_p, B_p).

(4) El bloque kernel genera los valores del kernel correspondientes a las coordenadas (F, C) que genera el bloque ctrl_submat.

(5) El bloque LeeGrab_AddrRam utiliza las coordenadas del píxel que se procesa (bloque cont_pxl) o de la submatriz de la imagen (bloque sub_mat) para generar la dirección de la memoria RAM donde se lee o guarda el píxel procesado.

(6) El bloque oper se encarga de hacer las operaciones ya sea de erosión o de dilatación y de ir sacando el área de la imagen que resulta después de una operación morfológica de apertura.

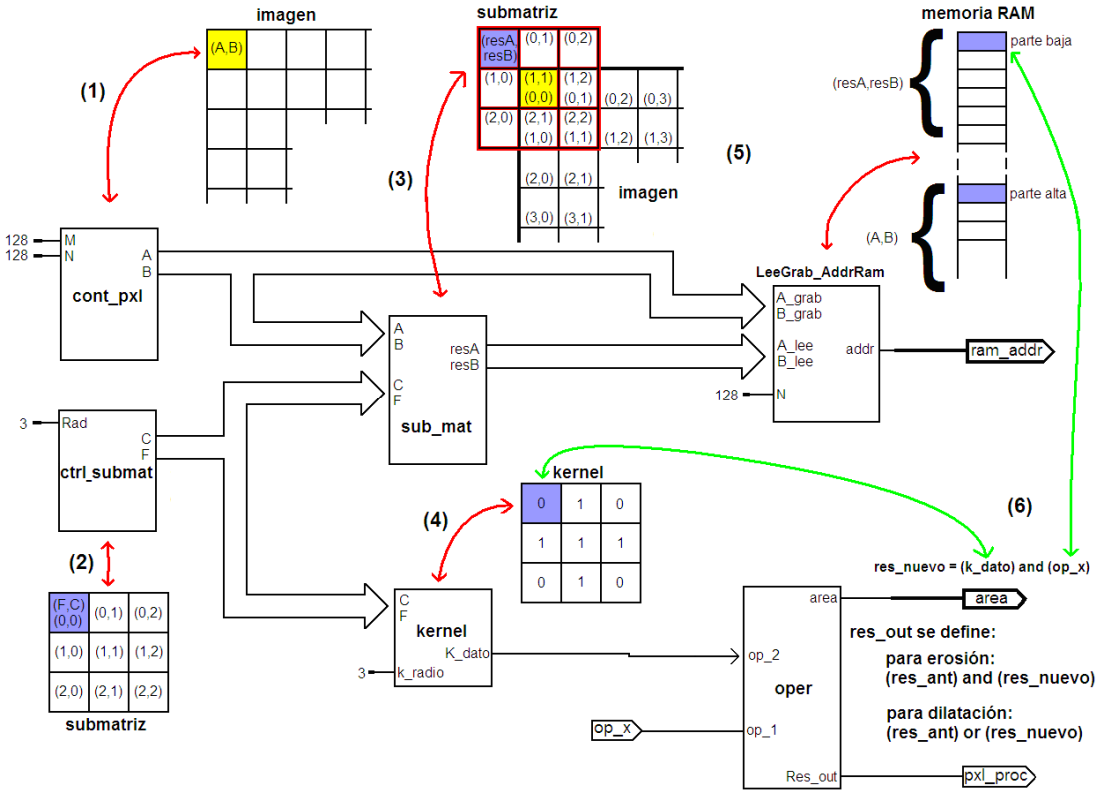

Figura 4.20. Diagrama simplificado del bloque de procesamiento.

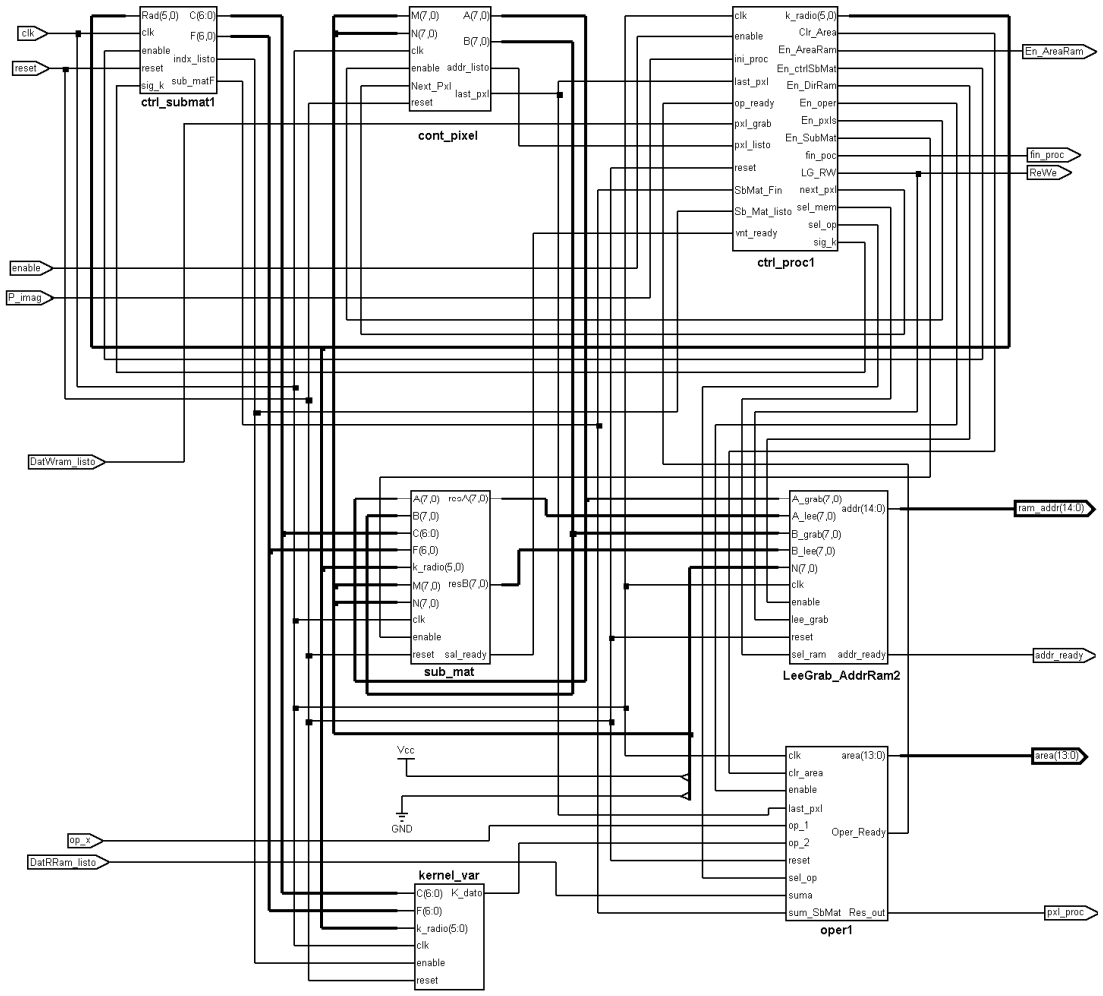

Figura 4.21. Diagrama completo del bloque de procesamiento.

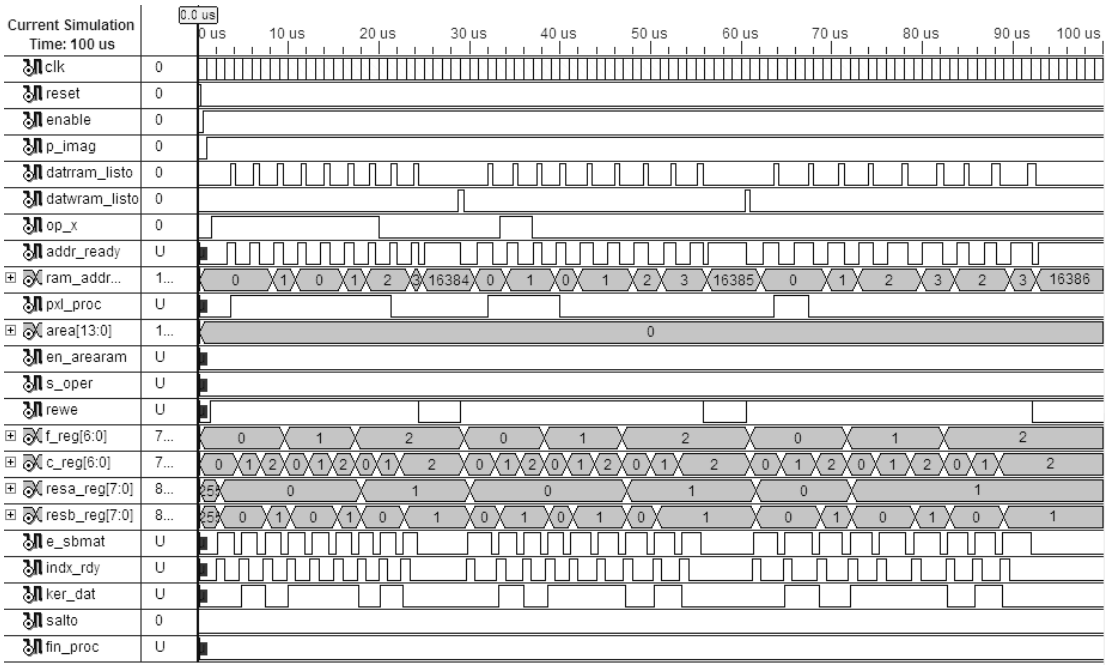

Figura 4.22. Simulación del bloque de procesamiento, se procesa un píxel con un kernel de 3x3.

4.08 Bloque area_block1.

Hace las restas de las áreas resultantes entre aperturas y guarda el resultado en una memoria RAM, cuando termina el procesamiento de la imagen, todos los resultados guardados forman el vector de resultados, cada elemento del vector es de 15 bits.

Envía el vector a la PC en dos bytes, el primer byte contiene la parte alta de resultado (bits 14 al 8) y el segundo la parte baja (bits 7 al 0). El programa de matlab une estos dos bytes para obtener el resultado correcto.

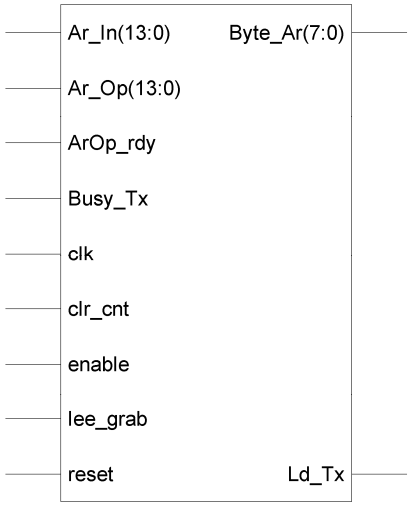

Figura 4.23. Bloque area_block1.

Ar_In – área de la imagen original.

Ar_Op – área de la imagen después de una apertura.

ArOp_rdy – indica que se tiene un dato nuevo en Ar_Op.

Busy_Tx – indica si el USART esta ocupado enviando un dato a la PC.

clr_cnt - inicializa el contador de direcciones de la memoria y comienza con el envío del vector de resultados.

lee_grab – indica si se va a grabar o leer la memoria.

Byte_Ar – byte que contiene el área guardada. Se envían dos bytes por resultado.

Ld_Tx – carga el byte de área en el USART para ser enviado.

Esta formado por tres bloques:

4.08.1 Bloque area_pecstrum.

Hace las restas de las áreas entre aperturas, guarda el resultado en memoria, lee la memoria para enviar el vector a la PC.

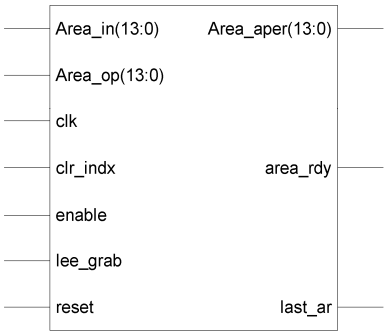

Figura 4.24. Bloque area_pecstrum.

Area_in - área de la imagen original.

Area_Op – área de la imagen después de una apertura.

clr_indx – inicializa el contador de direcciones de la memoria.

lee_grab – indica si lee o graba la memoria.

Area_aper – cuando lee_grab es 1, Area_aper se utiliza para sacar el vector de resultados guardado en la memoria.

area_rdy – indica que en Area_aper se tiene una salida lista para ser enviada.

last_ar – indica que el dato que tiene Area_aper es el último del vector de resultados.

4.08.2 Bloque send_area.

Envía cada elemento del vector de resultados en dos bytes a la PC, genera las señales necesarias para el USART.

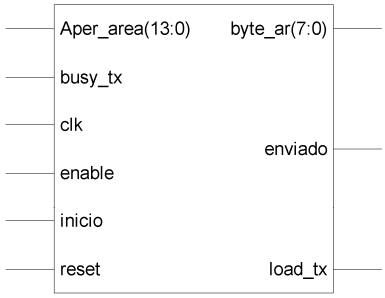

Figura 4.25. Bloque send_area.

Aper_area – resultado que se envía a la PC (elemento del vector de resultados).
busy_tx – señal que indica si el USART esta ocupado.
inicio – señal que indica cuando comenzar con el envió del vector.
byte_ar – byte a enviar por el USART.
enviado – indica que ya se envío un elemento del vector de resultados (14 bits).
load_tx – carga byte_ar en el USART para enviarlo a la PC.

4.08.3 Bloque ctrl_area1.

Controla los tiempos de las señales para la lectura y escritura de la memoria del bloque area_pecstrum, así como para el envío del vector a la PC.

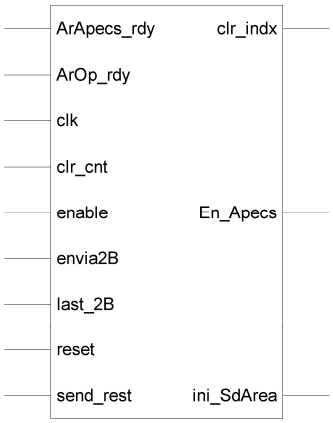

Figura 4.26. Bloque ctrl_area1.

ArApecs_rdy – para saber si en Area_aper se tiene una salida lista para ser enviada.
ArOp_rdy – indica que se tiene una nueva área en Ar_Op.
Clr_cnt – inicializa el contador de direcciones de la memoria y comienza con el envío del vector de resultados.
envia2B – indica que ya se enviaron 2 bytes correspondientes a un elemento del vector de resultados.

last_2B – indica que se esta enviando el ultimo elemento del vector de resultados.

send_rest – indica si se configuran los bloques para enviar o para hacer las restas de las áreas.

clr_indx – inicializa el contador de direcciones de la memoria.

En_Apecs – habilita el bloque area_pecstrum

ini_SdArea – se usa para iniciar el envío del vector de resultados.

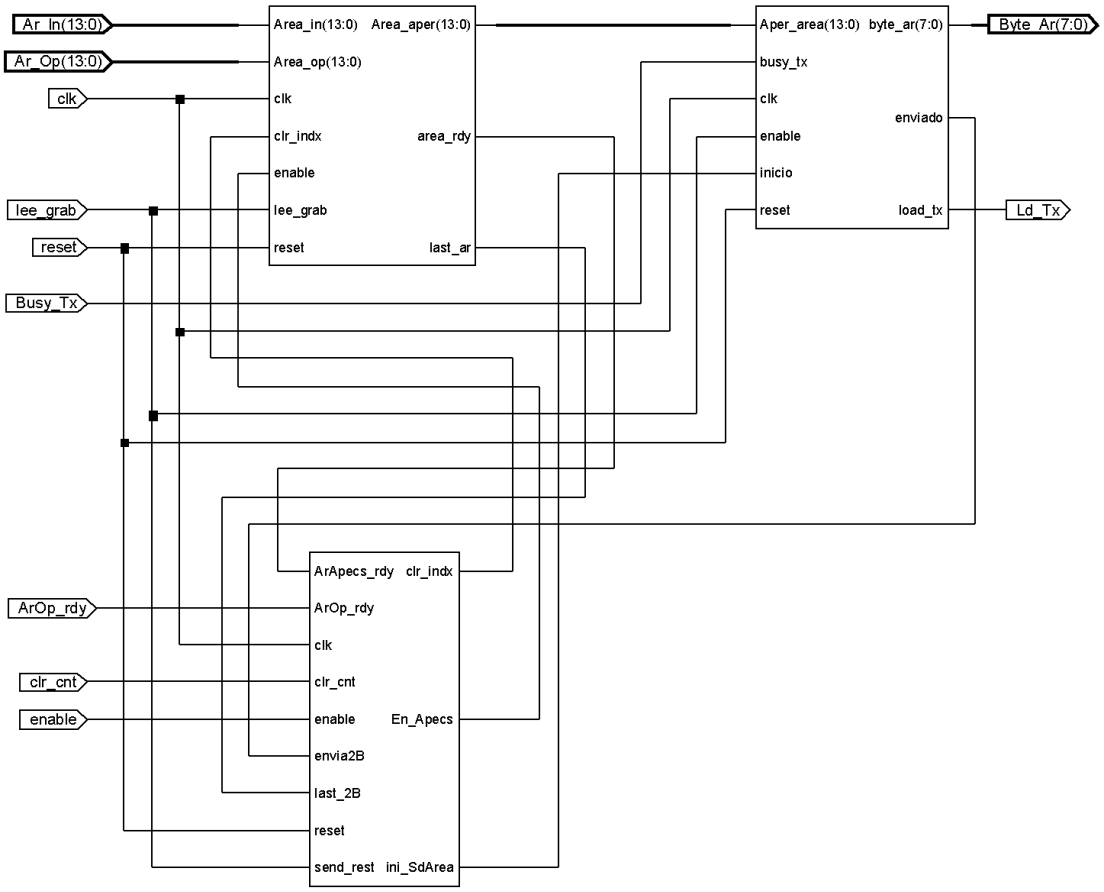

Figura 4.27. Diagrama de conexión de los bloques area_pecstrum, ctrl_area1 y send_area.

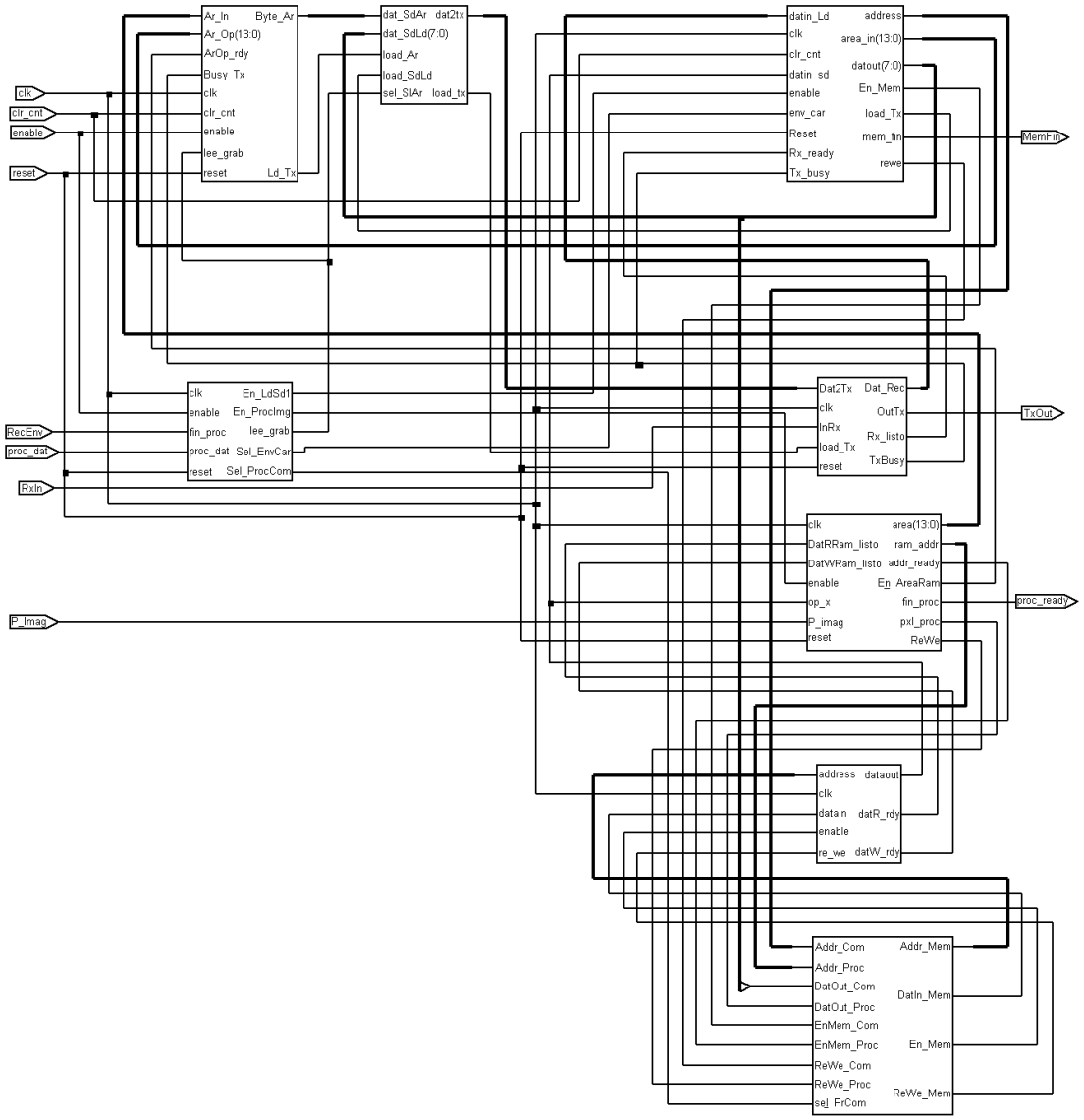

Figura 4.28. Diagrama completo del sistema de procesamiento.

4.09 Interfaz grafica de usuario.

Se hizo una interfaz grafica de usuario con matlab, para abrir una imagen, en formato jpg o bmp, y acondicionarla para enviarla al FPGA; así como para recibir la imagen procesada y el vector de resultados. El acondicionamiento consiste en cambiar de tamaño la imagen y convertirla a blanco y negro.

Conforme la tesis se iba desarrollando se necesitó comparar los resultados de las operaciones de erosión y dilatación primero por separado y posteriormente en una apertura (una erosión seguida de una dilatación), con el fin de verificar que los resultados que se obtenían con el FPGA eran los mismos que los de matlab.

4.09.1 Operaciones que realiza la interfaz.

La interfaz utiliza las siguientes instrucciones para realizar las operaciones morfológicas:

Imopen:

imag_2 = imopen(imag_1, NHOOD)

Realiza la apertura de la imagen binaria IM con el elemento estructurante NHOOD, que es un arreglo de unos y ceros que especifican la vecindad del elemento estructurante.

Imclose:

imag_2 = imclose(imag_1, NHOOD)

Realiza la cerradura de la imagen binaria IM con el elemento estructurante NHOOD, que es un arreglo de unos y ceros que especifican la vecindad del elemento estructurante.

Imerode:

imag_2 = imerode(imag_1, NHOOD)

Erosiona la imagen IM, donde NHOOD es un arreglo de unos y ceros que especifican la vecindad del elemento estructurante.

Imdilate:

imag_2 = imdilate(imag_1, NHOOD)

Dilata la imagen IM utilizando el elemento estructurante NHOOD, que es una matriz de unos y ceros que especifican la vecindad del elemento estructurante.

El pecstrum utiliza la instrucción imopen sucesivamente con el kernel creciente.

El kernel se generó con la instrucción strel('disk', R,0), esta instrucción crea una matriz de (2R+1 x 2R+1), y genera el circulo con todos los pixeles cuya distancia al centro de la matriz es menor que R.

Área de la imagen.

Para obtener el área de la imagen, PECS_1 suma todos los pixeles blancos de la imagen.

4.09.2 Envío y recepción de datos.

Se utiliza un puerto serial para el intercambio de datos con el FPGA.

Para que matlab pueda utilizar el puerto serial de la PC primero se debe configurar, establecer cual puerto se va utilizar, la velocidad de transmisión, tamaño de los buffers de entrada y salida.

El puerto serial se configuro a 9.6kbps

```
s = serial('COM1');
s.BaudRate=9600; % se configura la velocidad a 9600.
s.StopBits=1; %se configura bit de paro a uno.
s.OutputBufferSize=100000; %tamaño de el buffer de salida.
s.InputBufferSize=100000; %tamaño de el buffer de entrada.
fopen(s); %se abre el puerto serial.
```

Para leer datos se usa la instrucción:

```
variable = fread(s,N_elementos); %lee el N_elementos.
```

Para enviar los datos se usa la instrucción:

fwrite(s,pxl); %envía pxl

Una vez utilizado el puerto se cierra

fclose(s);

delete(s);

clear s;

La imagen se envía por filas siguiendo la secuencia que se muestra en la tabla 4-6.

(A,B) (0,0) 1	(0,1) 2	(0,2) 3	. . .	(0,127) 128
(1,0) 129	(1,1) 130	(1,2) 131	. . .	(1,127) 256
(2,0) 257	(2,1) 258	(2,2) 259		(2,127) 384
(127,0) 16257	(127,1) 16258	(127,2) 16259		(127,127) 16384

Tabla 4-6. Orden en que se envían los elementos de la imagen.

El primer elemento que se envía es el (0,0), el segundo es el (0,1) y así sucesivamente.

De la misma manera cuando pecs_1 recibe la imagen, el primer dato que recibe es el (0,0) el programa se encarga de guardarlos en forma de matriz para mostrar la imagen.

Cada elemento del vector de resultados esta formado por dos bytes, el programa se encarga de unirlos para obtener los valores correctos.

Si el pecstrum ejecuta 15 aperturas, el vector de resultados debe ser de 15 elementos, el FPGA envía dos bytes por resultado (30 bytes), el

programa los convierte a binario, los concatena y los regresa a decimal para obtener el valor correcto.

4.09.3 Descripción de la interfaz.

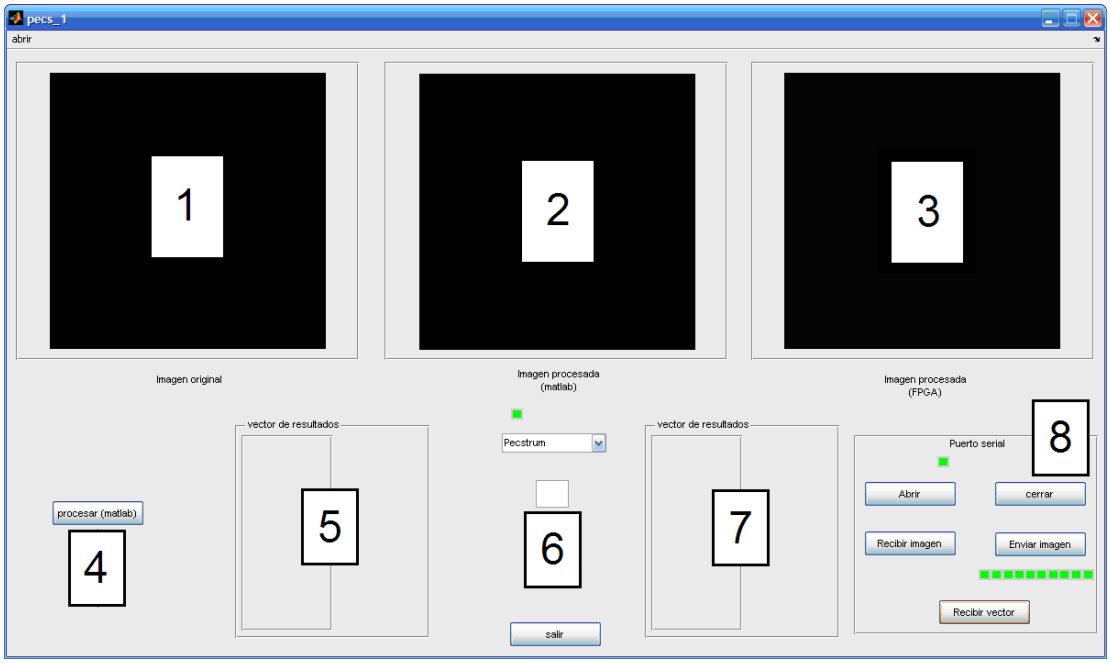

Figura 4.29. Descripción de la interfaz de usuario.

(1) Muestra la imagen original.

(2) Muestra la imagen procesada con matlab.

(3) Muestra la imagen procesada con el FPGA.

(4) Botón para procesar la imagen con matlab.

(5) Vector de resultados obtenidos con matlab.

(6) Selección del tipo de operación y el tamaño máximo del kernel.

(7) Vector de resultados obtenidos con el FPGA.

(8) Control de puerto serial. Abrir y cerrar puerto; enviar y recibir imagen; recibir vector de resultados.

Capitulo 5.

Resultados

Resultados.

La puesta en espera del sistema se lleva a cabo a través del interruptor SW0, el botón BTN0 resetea el sistema.

Los interruptores sw1 ysw2 controlan el funcionamiento del programa:

SW7	SW6	Función
0	0	Cargar imagen
0	1	Procesar imagen
1	0	Enviar imagen procesada
1	1	Enviar vector de resultados

Para cargar la imagen en memoria los interruptores deben estar en cero, y se pulsa el botón BTN3 para iniciar la carga. Una vez cargada la imagen y tras configurar los interruptores SW7 y SW6 (0,1) el sistema comenzará el procesamiento una vez pulsado el botón BTN2. El usuario puede elegir entre enviar la imagen resultante o el vector de resultados configurando los interruptores (1,0) o (1,1) y pulsando el botón BTN3. El sistema utiliza el led LD0 de la tarjeta para indicar carga completa de la imagen, envío completo de la imagen y procesado completo de la imagen.

La interfaz grafica de matlab abre una imagen en formato jpg o bmp, y las convierte en imágenes binarias para enviarlas al FPGA, tiene la opción de procesar la imagen utilizando los comandos de matlab strel('disk',radio,0) para generar el elemento estructurante e imopen para la apertura, lo que permite comparar las graficas que entrega el FPGA con lo que se obtiene con matlab.

Las primeras pruebas que se hicieron, consistieron en procesar la imagen un determinado número de veces en el FPGA y comparar con los

resultados que se obtienen de matlab para ver si ambas imágenes eran iguales.

Las figuras 5.1 a 5.4 muestran los resultados parciales del pecstrum para imágenes a las que se les realizaron 11 aperturas con el elemento estructurante creciente, a simple vista se puede notar que las imágenes resultantes son iguales a las que se obtienen con matlab. Se compararon píxel a píxel las imágenes resultantes y se comprobó que eran iguales.

El vector de resultados obtenido con el programa del FPGA es de la misma longitud y presenta los mismos valores que los obtenidos con las instrucciones de matlab.

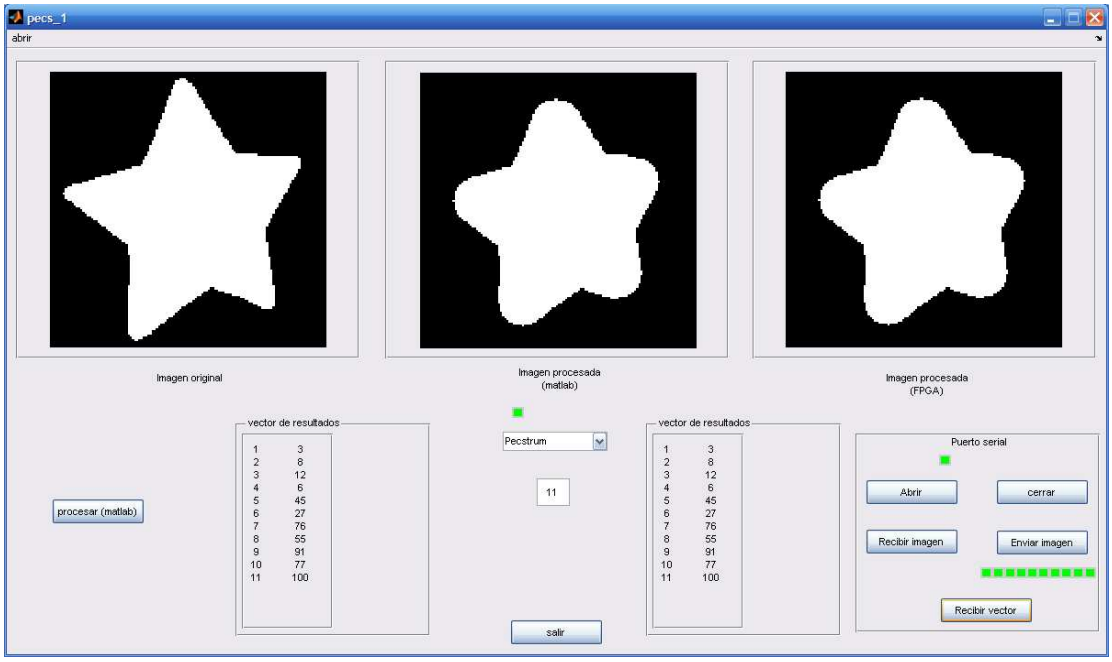

Figura 5.1. Procesamiento parcial del pecstrum sobre la imagen de una estrella con elemento estructurante máximo de 23x23

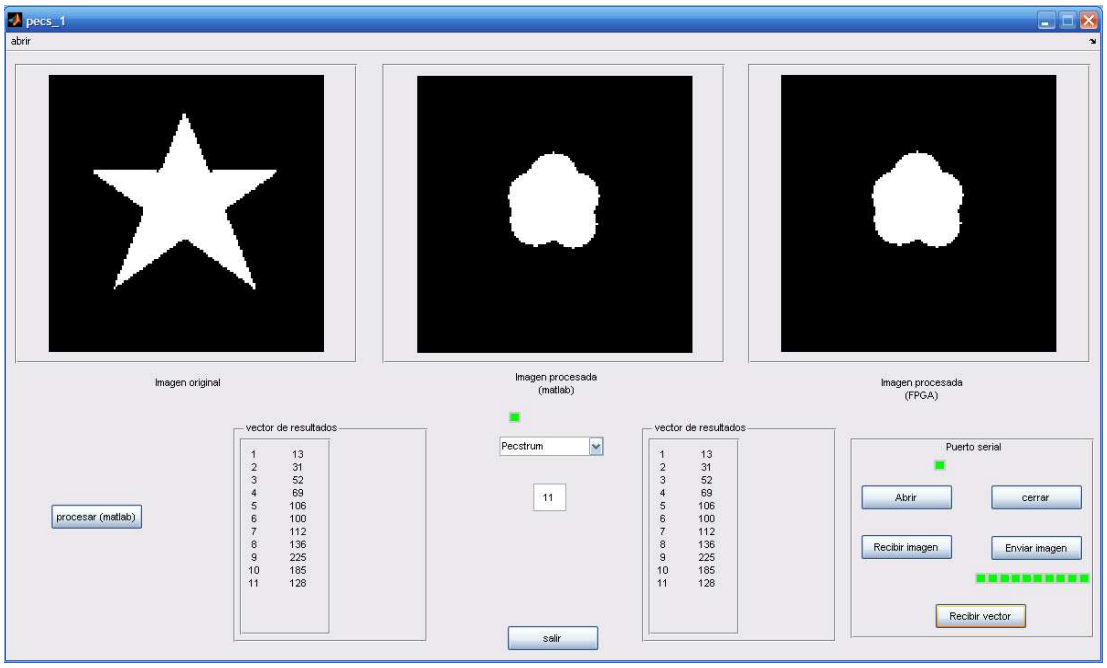

Figura 5.2. Procesamiento parcial del pecstrum sobre la imagen una estrella con elemento estructurante máximo de 23x23

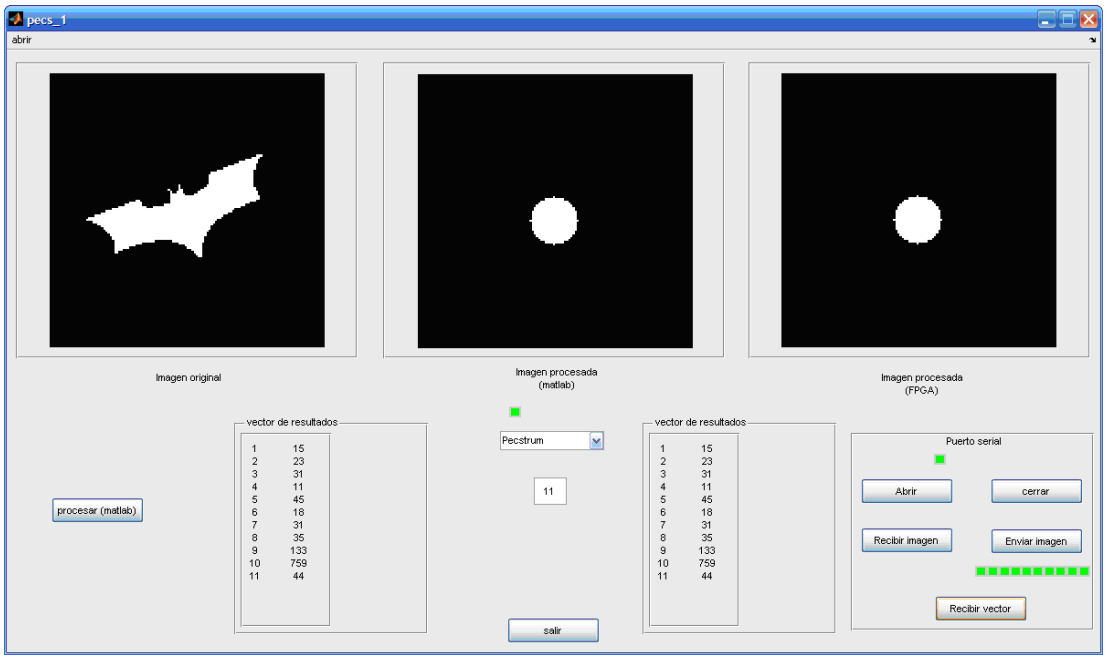

Figura 5.3. Procesamiento parcial del pecstrum sobre la imagen de un murciélago con elemento estructurante máximo de 23x23

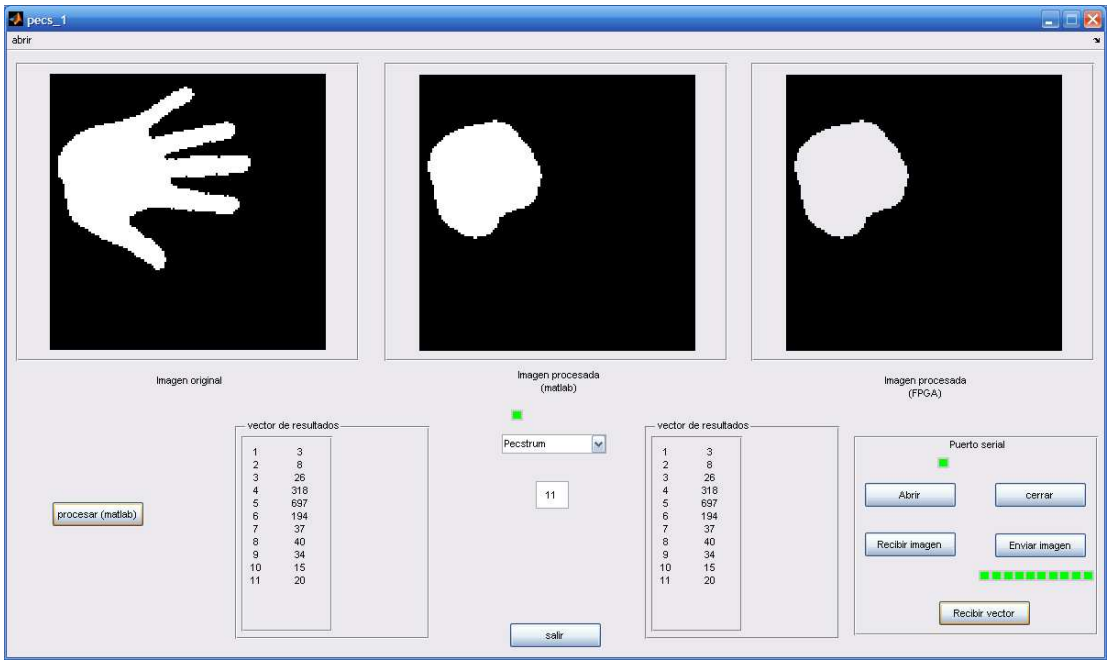

Figura 5.4. Procesamiento parcial del pecstrum sobre la imagen de una mano con elemento estructurante máximo de 23x23

La instrucción strel('disk',radio,0) genera un circulo de diámetro 2*radio+1, para 11 aperturas el kernel generado de esta manera tiene los siguientes tamaños: 3x3, 5x5, 7x7, 9x9, 11x11, 13x13, 15x15, 17x17, 19x19, 21x21, 23x23.

De las figuras 5.1 a 5.4 se observa que el tiempo de procesamiento va a depender del tamaño de la imagen que se procesa, aunque a las tres se les aplicaron 11 aperturas, el área de las imágenes procesadas es diferente en cada una.

La figura 5.5 tiene el procesamiento de un circulo de radio 10 obtenido con la instrucción de matlab strel('disk', 10,0), esta instrucción genera un circulo de diámetro 21 pixeles. En este ejemplo se realizaron 10 aperturas para reducir el área de la imagen a cero, el tamaño del kernel que se utilizo en la última apertura fue de 21x21. Los valores que se obtienen con matlab y con el fpga son iguales.

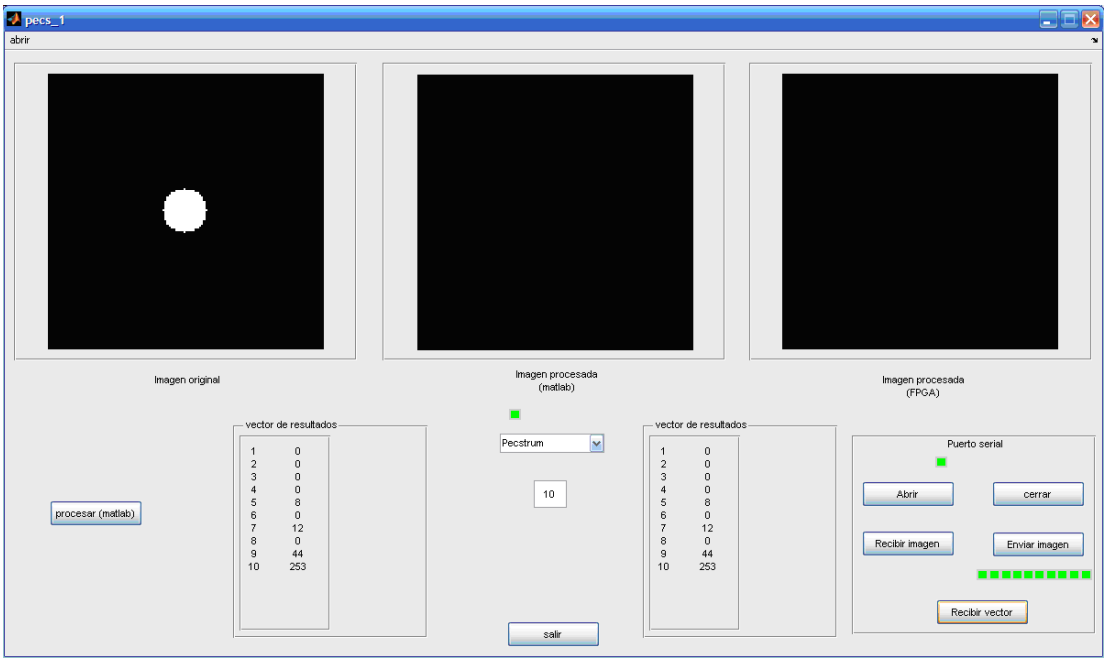

Figura 5.5. Operador pecstrum aplicado a un circulo de diámetro 21.

Las figuras 5.6 y 5.7 muestran los resultados del operador pecstrum sobre las imágenes de un murciélago y una mano respectivamente. La imagen del murciélago se proceso en 12 aperturas. La imagen de la mano se proceso en 23 y se realizo otra apertura para comprobar que ya no se obtienen más valores.

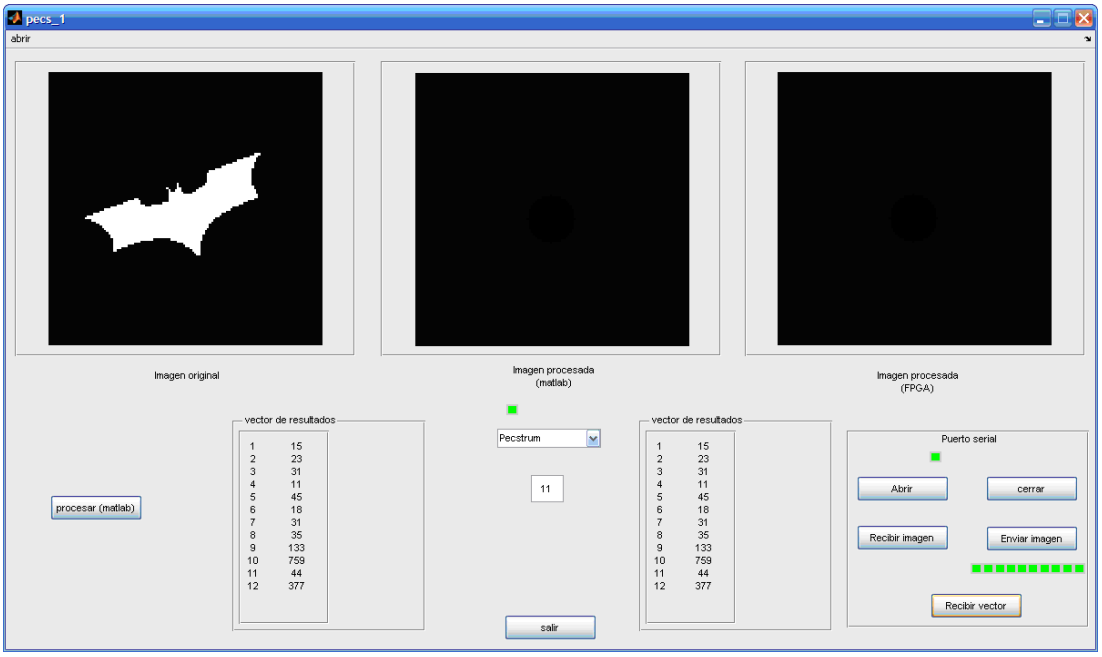

Figura 5.6. Operador pecstrum aplicado a una imagen binaria.

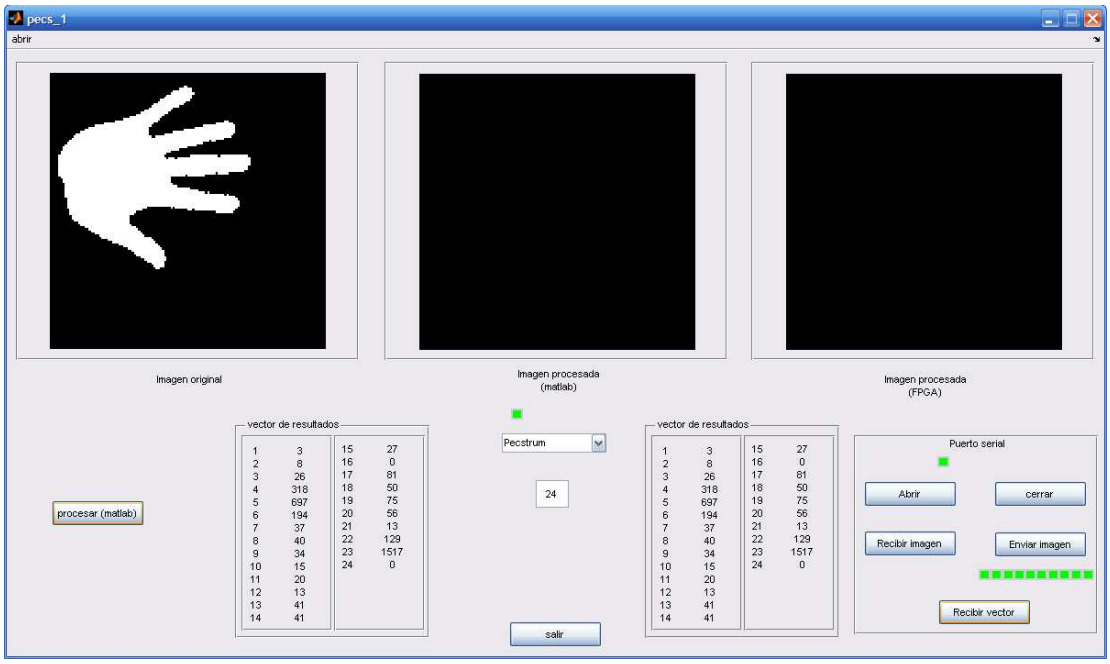

Figura 5.7. Operador pecstrum aplicado a la imagen de una mano, área en pixeles 3435.

Figura 5.8. Operador pecstrum aplicado a una imagen binaria, área en pixeles 3435.

El área de la mano en la figura 5.7 es de 3435 pixeles. En la figura 5.8 se procesa una imagen con la misma área pero con diferente forma, para procesar ésta imagen sólo se utilizaron 12 aperturas, a diferencia de las 24

de la imagen de la mano. Los resultados del pecstrum entregan información sobre la forma de la imagen, no tanto del área.

El bloque de procesamiento resulto más lento que matlab, la imagen de la figura 5.7 se proceso en 3.12 minutos en el FPGA, mientras que con matlab fue de 12s con 24 aperturas. Sin embargo los recursos utilizados del FPGA son mínimos, en la figura 5.9 se muestran los recursos utilizados por todo el circuito, como se puede observar, la mayoría es menor al 10%.

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	393	9,312	4%	
Number of 4 input LUTs	668	9,312	7%	
Logic Distribution				
Number of occupied Slices	458	4,656	9%	
Number of Slices containing only related logic	458	458	100%	
Number of Slices containing unrelated logic	0	458	0%	
Total Number of 4 input LUTs	705	9,312	7%	
Number used as logic	640			
Number used as a route-thru	37			
Number used for 32x1 RAMs	28			
Number of bonded IOBs				
Number of bonded	17	232	7%	
Number of RAMB16s	2	20	10%	
Number of BUFGMUXs	1	24	4%	
Number of MULT18X18SIOs	3	20	15%	

Figura 5.9. Recursos ocupados por el programa completo.

En la figura 5.10 se muestran los recursos utilizados únicamente por el circuito de procesamiento. Estos resultaron ser poco menos de la mitad de los recursos que utiliza el bloque completo.

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	172	9,312	1%	
Number of 4 input LUTs	361	9,312	3%	
Logic Distribution				
Number of occupied Slices	221	4,656	4%	
Number of Slices containing only related logic	221	221	100%	
Number of Slices containing unrelated logic	0	221	0%	
Total Number of 4 input LUTs	363	9,312	3%	
Number used as logic	361			
Number used as a route-thru	2			
Number of bonded IOBs				
Number of bonded	41	232	17%	
Number of BUFGMUXs	1	24	4%	
Number of MULT18X18SIOs	3	20	15%	

Figura 5.10. Recursos del FGPA utilizados por el bloque de procesamiento.

Se puede aprovechar el procesamiento en paralelo que puede realizarse con el FPGA, e implementar más circuitos de procesamiento, dividir la imagen en varios bloques pequeños, y procesarlos en paralelo.

Trabajo a futuro.

Como trabajo a futuro se plantean algunas posibilidades de arquitecturas de diseño que podrían incrementar el aprovechamiento de la característica de trabajo en paralelo del FPGA.

(a)Procesamiento por filas completas:

Con la misma estructura del circuito que se presenta en esta tesis y modificando algunos bloques se puede trabajar por filas completas. Para un kernel de 3x3 el programa debe generar las direcciones de la fila completa de la submatriz, leer los valores de la memoria y al mismo tiempo generar la fila correspondiente del kernel.

Fila submatriz llena con los valores de la imagen

subM0	subM1	subM2

Fila kernel

K0	K1	K2

Por ultimo realiza la operación.

Para erosión:

Res_Op=(subM0 and k0) and (subM1 and k1) and (subM2 and k2)

Si al menos una operación (subM and k) da 0 el resultado de Res_Op será 0.

Para dilatación:

Res_Op=(subM0 and k0) or (subM1 and k1) or (subM2 and k2)

Si al menos una operación (subM and k) da 1 el resultado de Res_Op será 1.

Al momento de hacer mas grande el kernel y la submatriz, para que se pueda procesar la fila completa, se necesita que el circuito que hace la operación se vaya modificando (haciendo mas grande), o que desde el inicio se defina la operación para el kernel de mayor tamaño con el que se va a trabajar.

Los elementos del kernel y de la imagen, por ser en blanco y negro, son unos y ceros y las filas pueden ser tratadas como vectores. Para un kernel máximo de 65 x 65, se estaría trabajando con vectores de 65 elementos para cualquier tamaño de kernel.

Se generan las direcciones inicial y final de la fila de la submatriz y se leen de la memoria RAM, el kernel también se genera por filas completas y se procesan en una sola operación.

Para el ejemplo del kernel de 3x3 se tiene:

Para la erosión, los elementos del vector que no corresponden a elementos de la submatriz se llenan con unos, el vector del kernel se llena de la misma manera.

$$Fila\_submat=SubM0,subM1,subM2,1,1,1\ldots,1$$

$$Fila\_Kernel=k0,k1,k2,1,1,1\ldots,1$$

La operación por filas queda

$$op=Fila\_submat \text{ and } fila\_Kernel$$

$$op=(subM0 \text{ and } k0),(subM1 \text{ and } k1),(subM2 \text{ and } k2),1,1,1\ldots,1$$

El resultado de la erosión se obtiene checando el valor de not(op), si es mayor a cero (lo que es equivalente a decir que al menos una operación subM and k es 0) entonces el resultado Res_Op es 0, si not(op) es cero, todas las operaciones intermedias fueron 1 y Res_Op es 1.

si not(op)>0 entonces Res_Op=0

si not(op)=0 entonces Res_Op=1

Para la dilatación, los elementos del vector que no corresponden a elementos de la submatriz se llenan con ceros.El vector del kernel se llena de la misma manera.

Fila_submat=SubM0,subM1,subM2,0,0,0…,0

Fila_Kernel=k0,k1,k2,0,0,0…,0

La operación por filas queda

op=Fila_submat and fila_Kernel

op=(subM0 and k0),(subM1 and k1),(subM2 and k2),0,0,0…,0

De esta manera para obtener el resultado de la dilatación, se checa el valor de op, si es mayor a cero (lo que es equivalente a decir que al menos una operación subM and k es 1) entonces el resultado Res_Op es 1, si op es cero, todas las operaciones intermedias fueron 0 y Res_Op es 0.

si op>0 entonces Res_Op=1

si op=0 entonces Res_Op=0

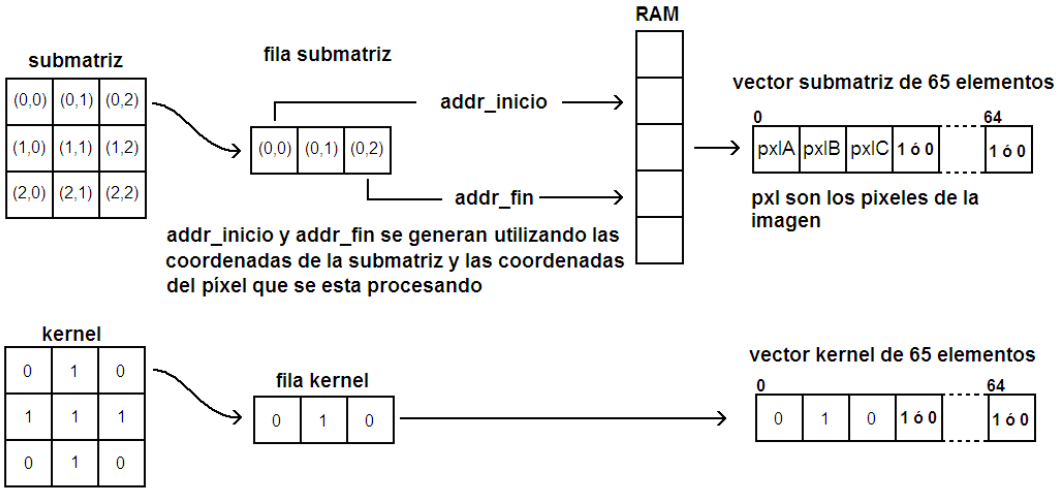

Figura 5.11. Generación del vector de la submatriz y del kernel

(b) Procesamiento de varios pixeles simultáneamente.

Se pueden procesar varios pixeles que estén sobre la misma fila de la imagen al mismo tiempo. La fila que se obtuvo para procesar el primer píxel se utiliza para procesar el segundo, solo se tiene que aumentar un píxel como se muestra en la figura 5.12.

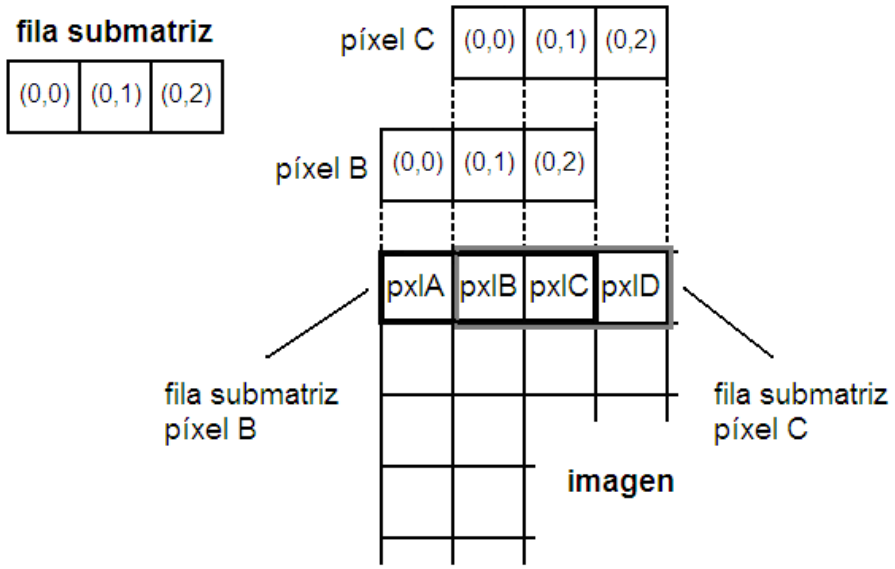

Figura 5.12. Filas para procesar los pixeles B y C

Para procesar 4 pixeles además de la fila para el primer píxel se deben leer tres pixeles adicionales.

Se genera la dirección addr_inicio con el primer elemento de la fila de la submatriz del primer píxel (píxel B en la figura 5.13) y addr_fin con el último elemento de la fila de la submatriz del último píxel (píxel E en la figura 5.13).

Se lee la memoria desde la dirección addr_inicio hasta addr_fin y se llenan los 4 vectores con los valores que les corresponde.

El vector del kernel solo se genera una vez, es el mismo para los 4 vectores.

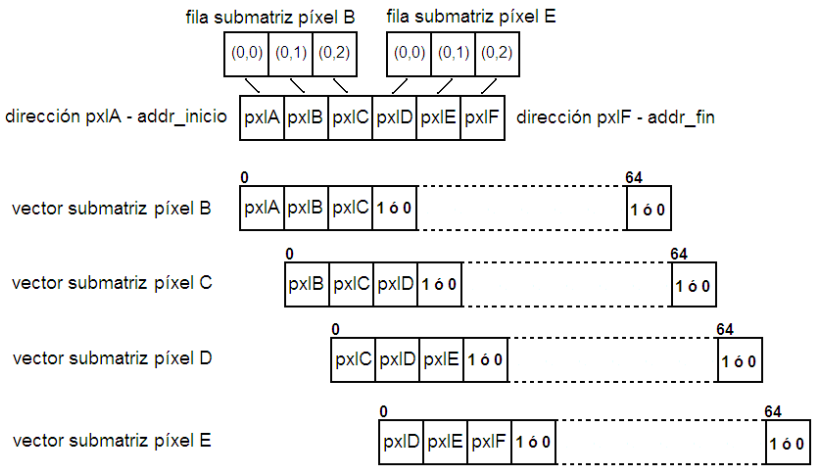

Figura 5.13. Generación de cuatro vectores.

Para procesar los 4 pixeles al mismo tiempo se necesitan 4 circuitos que realicen la operación AND con 2 vectores de 65 elementos.

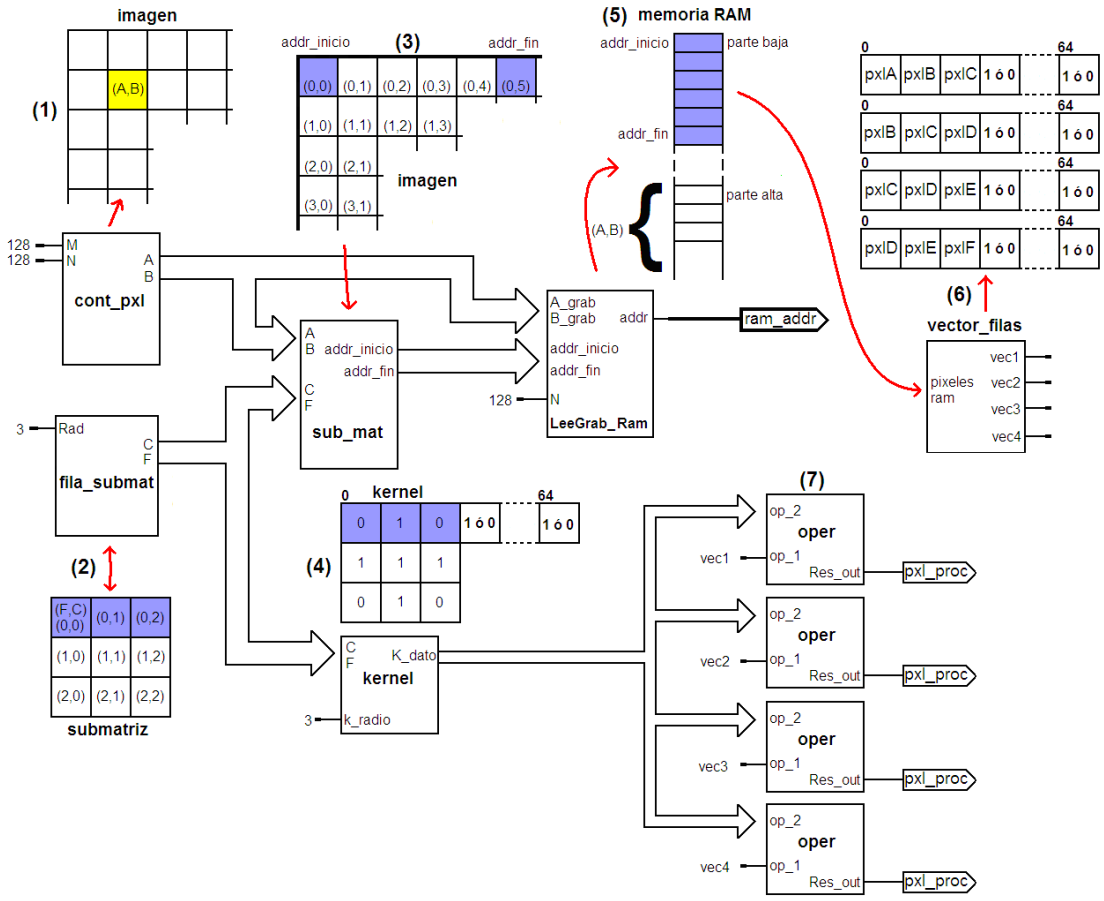

Figura 5.14. Diagrama simplificado para trabajar por filas y para procesar 4 pixeles a la vez.

(1) se generan las coordenadas del píxel que se va a procesar.

(2) Se genera la fila de la submatriz.

(3) Se generan las direcciones addr_inicio y addr_fin para leer los pixeles que se necesitan para llenar los vectores.

(4) Se genera el vector kernel.

(5) Se lee la memoria RAM

(6) Se forman los vectores de la submatriz llenos con los pixeles de la imagen.

(7) Se hacen las operaciones.

(c) Procesamiento de la imagen por secciones.

La imagen se divide en bloques o secciones rectangulares y se utiliza un circuito de procesamiento para cada bloque.

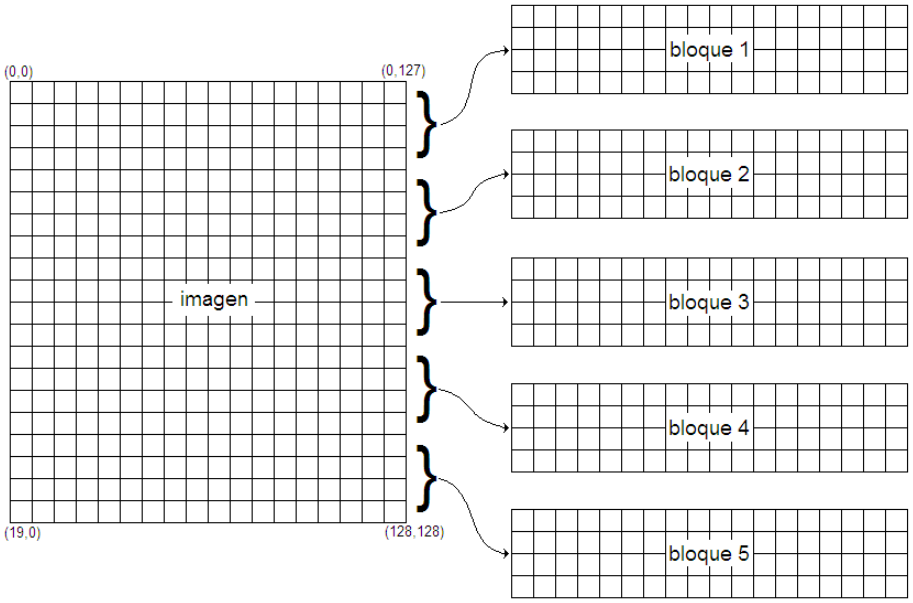

Figura 5.15. Ejemplo de imagen de 20x128 dividida en 5 bloques de 4x128.

Los circuitos de procesamiento pueden trabajar en paralelo, de esta manera se puede reducir el tiempo de procesamiento. Sin embargo, al ser una memoria única, los bloques de procesamiento no pueden acceder a la memoria al mismo tiempo. Los circuitos de procesamiento se modifican para sacar las direcciones de inicio y fin de lectura, estas señales se conectan a un bloque (LeeGrab_Ram modificado) que se encargara de leer el vector completo y de asignarlo al circuito de procesamiento que le corresponda.

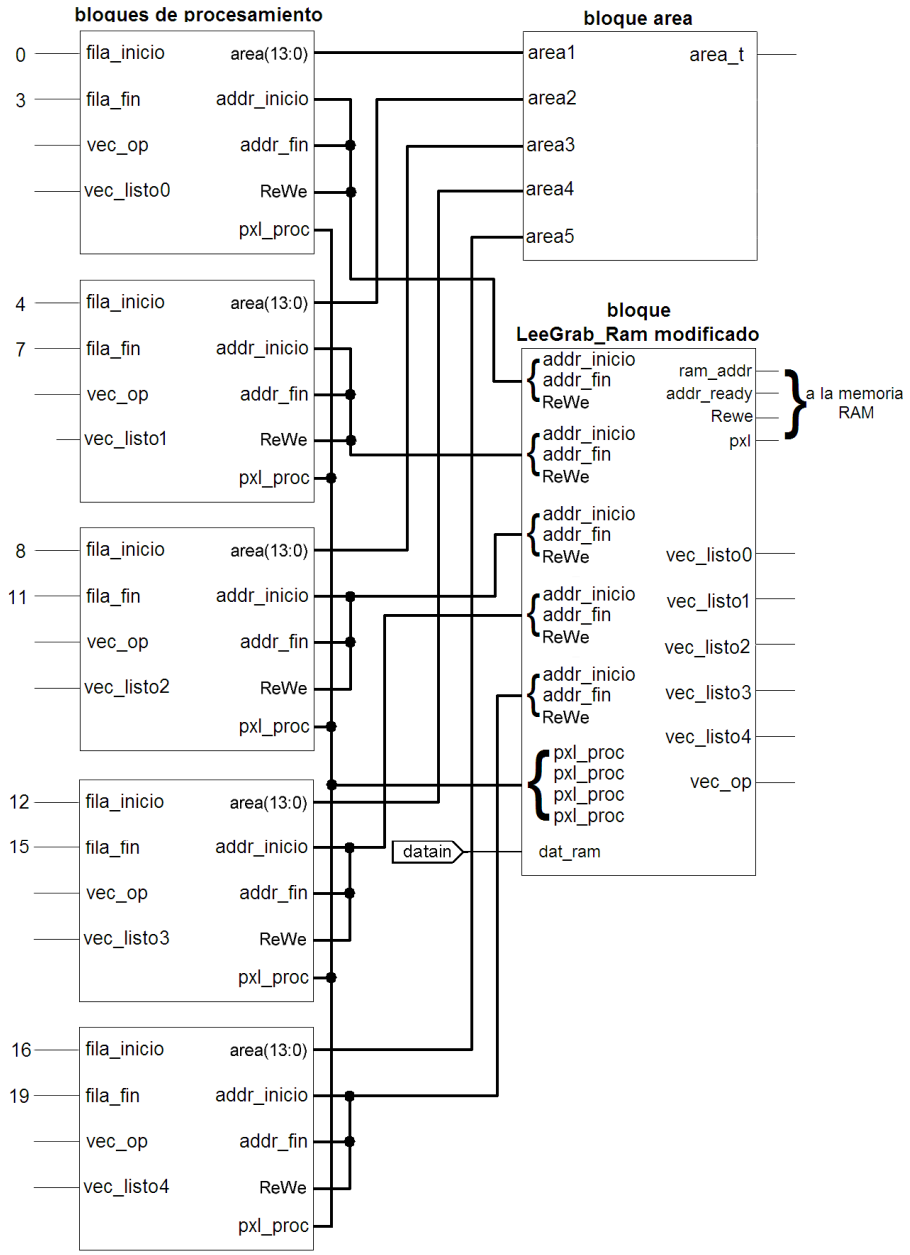

Figura 5.16. Diagrama simplificado para procesar la imagen dividida en bloques.

El bloque LeeGrab_Ram lee la memoria, forma el vector vec_op y lo asigna al circuito de procesamiento que le corresponda.

El número de bloques en el que se puede dividir la imagen depende de cuantos circuitos de procesamiento se puedan implementar en el FPGA.

Capitulo 6.

Conclusiones

Conclusiones.

Los FPGA, son una herramienta muy poderosa al momento de diseñar un circuito debido a que son reprogramables, lo que permite ir corrigiendo los errores que se presentan en cada etapa del diseño. El lenguaje de programación en ocasiones resulta difícil de comprender, más aun cuando se tratan de detectar errores que no resultan tan obvios de ver como los que se presentan cuando el programa no se escribe con la sintaxis apropiada.

El circuito cumplió con el objetivo procesar el operado pecstrum sobre una imagen binaria, y entrego los mismos resultados que matlab, aunque la velocidad de procesamiento fue menor que la de matlab, sin embargo, el circuito de procesamiento resultó bastante compacto (menor al 5%), y no se debe olvidar la ventaja que presentan los FPGAs al poder implementar varios circuitos de procesamiento y procesar la imagen por secciones mas pequeños en forma paralela, con lo que se conseguiría reducir el tiempo de procesamiento.

Bibliografía.

[1] Yörük, E., Dutagaci, H., Sankur, B. (2005). *Hand Based Biometry*. Proc. SPIE, Vol. 5685, 1106 (2005); doi:10.1117/12.587815.

[2] Zois, E.N., Anastassopoulos, V. (2009). *Modeling the pattern spectrum as a Markov process and its use for efficient shape classification*. Image Processing (ICIP), 2009 16th IEEE International Conference on.

[3] Ramirez Cortes, J. M., Gomez Gil, P., Sanchez Perez, G., Prieto-Castro, C. (2009). *Shape-based hand recognition approach using the morphological pattern spectrum*. Journal of Electronic Imaging 18(1), 013012 (Jan–Mar 2009).

[4] Kumara, A. et al. (2006). *Personal authentication using hand images*. Pattern Recognition Letters Volume 27 (2006) 1478-1486.

[5] Ramirez Cortes, J. M., Gomez Gil, P., Sanchez Perez, G. Baez Lopez, M. (2008). *A Feature Extraction Method Based on the Pattern Spectrum for Hand Shape Biometry*. Proceedings of the World Congress on Engineering and Computer Science 2008. ISBN: 978-988-98671-0-2.

[6] Pratt, William K. *Digital image processing, PIKS Scientific Inside*, cuarta edición. Editorial Wiley-Interscience.

[7] Gonzalez, Rafael C., Woods, Richard E. *Digital image processing*, segunda edición. Editorial Prentice Hall.

[8] Terissi, Lucas D., Cipollone, L., Baldino, P (2006). *Sistema de Reconocimiento de Iris*. Revista Argentina de Trabajos Estudiantiles Vol. I - Nº 2 - Marzo 2006.

[9] Brown, S., Vranesic Z. *Fundamentals of digital logic with VHDL design*, Segunda edición. Editorial Mc. Graw Hill.

[10] Pardo Carpio, F., Boluda Grau, Jose A. *VHDL lenguaje para síntesis y modelado de circuitos*. Editorial ra-ma.

[11] Chu, Pong P. *FPGA prototyping by VHDL Examples – xilinx Spartan-3 versión*. Editorial Wiley-Interscience.

[12] Deschamps, Jean-Pierre, Bioul, Géry Jean Antoine, Sutter, Gustavo D. *Synthesis of arithmetic circuits FPGA, ASIC, and Embedded Systems*. Editorial Wiley-Interscience.

[13] Pedroni Volnei A. *Circuit design with VHDL*. Editorial MIT Press.

[14] Spartan-3E FPGA Family: Data Sheet. Xilinx, notas de aplicación DS312.

Apéndice A. listado de los programas y simulaciones.

Listado bloque principal

```
----------------------------------------------------
--programa completo
--une los bloques de comunicación, área y procesamiento
----------------------------------------------------

library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity prog_completo is
generic (sz_dat : integer:=8;          --8 para una imagen de 128 x 128
      sz_indx: integer:=8;             --para submatrices de 128x128 max.
      sz_k: integer:=7;                --para kernel máximo de 64x64
      sz_rad: integer:=6;
      dim_ram : integer :=14);         --para imagen de 128x128 -> 16384
port(reset : in std_logic;
      enable : in std_logic;
      clk : in std_logic;
      RxIn : in std_logic;
      clr_cnt : in std_logic;
      P_Imag : in std_logic;
      proc_dat : in std_logic;  --para selección
      RecEnv : in std_logic;    --para selección
      MemFin : out std_logic;
      proc_ready: out std_logic;
      led : out std_logic;
      ve_dato  : out std_logic;
      ve_dato2 : out std_logic;
      ve_dato3 : out std_logic;
      ve_dato4 : out std_logic;
      ve_dato5 : out std_logic;
      TxOut : out std_logic
      );
end prog_completo;

architecture BEHAVIORAL of prog_completo is

--com_usart
Signal TxBusy1 : Std_Logic;
signal Rx_listo1: Std_Logic;
signal Dat_Rec1 : Std_Logic_Vector(7 downto 0);
--imag
Signal Area_proc : std_logic_vector (dim_ram-1 downto 0);  -- para leer
signal Ram_Addr1 : std_logic_vector (dim_ram downto 0);    -- para escribir
signal addr_ready1 : std_logic;
signal En_AreaRam1 : std_logic;
signal pxl_proc1 : std_logic;
signal ReWe_proc : std_logic;
--load_send1
Signal datout1 : std_logic_vector(sz_dat-1 downto 0);
signal area_in1: std_logic_vector(dim_ram-1 downto 0);
signal ReWe_LdSd : std_logic;
signal EnMem_LdSd : std_logic;
signal addr_LdSd: std_logic_vector(dim_ram downto 0);
signal load_Tx1: Std_Logic;
```

```
signal Mem_Fin1: Std_Logic;
--ctrl3b
Signal En_LdSd11 : std_logic;
Signal Sel_EnvCar1 : std_logic;
signal Sel_ProcCom1 : std_logic;
signal En_ProcImg1 : std_logic;
signal le_gr : std_logic;
--ram_1b
Signal Dat2Proc  : std_logic;
signal datR_rdy1 : std_logic;
signal datW_rdy1 : std_logic;
--muxs
Signal En_Mem1 : std_logic;
signal Addr_Mem1 : std_logic_vector (dim_ram downto 0);
signal DatIn_Mem1: std_Logic;
signal ReWe_Mem1 : std_Logic;
--area_block1
Signal byte_area : std_logic_vector(7 downto 0);    -- byte a transmitir
signal Ld_Tx : std_logic;
--mux_area
Signal dat2_Tx : std_logic_vector(7 downto 0);
signal load_Tx : std_logic;
-------------------------------------------------
--componentes.
-------------------------------------------------
component com_usart is
       generic (sz_dat: integer:=8;
              dim_ram : integer :=14);
   port (reset   : in std_logic;
              Clk     : in std_logic;
              InRx    : in std_logic;
              OutTx   : out std_logic;
              load_Tx : in std_logic;
              Dat2Tx  : in  Std_Logic_Vector(7 downto 0);
              TxBusy  : out Std_Logic;
              Rx_listo: out Std_Logic;                      -- Byte listo
              Dat_Rec : out Std_Logic_Vector(7 downto 0)    -- Byte recibido
              );
end component com_usart;

component imag is
       generic (sz_indx: integer:=8;
              dim_ram : integer :=14;
              size_k: integer:=7;
              sz_rad: integer:=6);
       port (clk : in std_logic;
              reset : in std_logic;
              enable : in std_logic;
              P_imag : in std_logic;
              DatRRam_listo : in std_logic;    --dato ram listo - leer
              DatWRam_listo : in std_logic;    --dato ram listo - grabar.
              op_x : in std_logic;
              addr_ready : out std_logic;
              ram_addr : out std_logic_vector (dim_ram downto 0);
              pxl_proc : out std_logic;
              Area : out std_logic_vector (dim_ram-1 downto 0);
              En_AreaRam:out std_logic;
              ReWe : out std_logic;
              fin_proc : out std_logic);       --ya se proceso la imagen
end component imag;
```

```
component load_send1 is
       generic (    sz_dat : integer := 8;
              dim_mem : integer := 14);  -- 13 bits para 4096 -> 64x64
                                         -- 14 bits para 16383 -> 128x128
       port (Clk : in      Std_Logic;
              Reset : in   Std_Logic;
              enable : in  Std_Logic;
              clr_cnt : in std_logic;
              env_car : in std_logic;
              datin_Ld : in std_logic_vector(sz_dat-1 downto 0);   --del usart
              datin_Sd : in std_logic;         --de la memoria de salida
              datout : out std_logic_vector(sz_dat-1 downto 0);
              area_in : out std_logic_vector(dim_mem-1 downto 0);
              --memoria
              re_we  : out std_logic;
              En_Mem : out std_logic;
              address: out std_logic_vector(dim_mem downto 0);
              mem_fin : out std_logic;
              --para enviar
              Tx_busy : in Std_Logic;
              load_Tx : out Std_Logic;
              --para recibir
              Rx_ready : inStd_Logic                        --Byte available
              );
end component load_send1;

component ctrl3b is
       port ( clk : in  std_logic;
              enable : in  std_logic;
              reset : in  std_logic;
              proc_dat : in  std_logic;
              fin_proc : in  std_logic;
              En_LdSd1 : out std_logic;
              Sel_EnvCar : out std_logic;
              Sel_ProcCom : out std_logic;
              lee_grab : out std_logic;
              En_ProcImg : out std_logic
              );
end component ctrl3b;

component ram_1b is
       generic (dim_ram : integer :=14);   --14->15bits imagen de 128x128x2
                                           --13->14bits imagen de 64x64x2
       port ( clk   : in  std_logic;
              enable: in  std_logic;
              re_we : in  std_logic;
              address : in  std_logic_vector(dim_ram downto 0);
              datain : in  std_logic;
              dataout : out std_logic;
              datR_rdy : out std_logic;
              datW_rdy : out std_logic
              );
end component ram_1b;

component muxs is
       generic (    dim_ram : integer :=14);
       port ( sel_PrCom : in std_logic;
              EnMem_Proc : in std_logic;
              Addr_Proc : in std_logic_vector (dim_ram downto 0);
              DatOut_Proc : in std_Logic;
              ReWe_Proc : in std_Logic;
```

```
              EnMem_Com : in std_logic;
              Addr_Com : in std_logic_vector (dim_ram downto 0);
              DatOut_Com : in std_Logic;
              ReWe_Com : in std_Logic;
              En_Mem : out std_logic;
              Addr_Mem : out std_logic_vector (dim_ram downto 0);
              DatIn_Mem : out std_Logic;
              ReWe_Mem : out std_Logic
              );
end component muxs;

component area_block1 is
       generic (sz_area : integer:=14;
              byte : integer := 8);
       Port ( clk : in STD_LOGIC;
              reset : in STD_LOGIC;
              enable : in STD_LOGIC;
              Ar_In : in STD_LOGIC_VECTOR (sz_area-1 downto 0);
              Ar_Op : in STD_LOGIC_VECTOR (sz_area-1 downto 0);
              ArOp_rdy : in STD_LOGIC;
              clr_cnt : in STD_LOGIC;
              lee_grab : in STD_LOGIC;
              Busy_Tx : in STD_LOGIC;
              Byte_Ar : out  STD_LOGIC_VECTOR (byte-1 downto 0);
              Ld_Tx : out  STD_LOGIC
              );
end component area_block1;

component mux_area is
       generic (    sz_dat : integer :=7);
       Port ( sel_SlAr : in std_logic;

              load_SdLd : in std_logic;
              dat_SdLd : in std_logic_vector (sz_dat downto 0);

              load_Ar : in std_logic;
              dat_SdAr : in std_logic_vector (sz_dat downto 0);

              load_tx : out std_logic;
              dat2tx : out std_logic_vector (sz_dat downto 0)
              );
end component mux_area;

-------------------------------------------------
Termina declaración de componentes
-------------------------------------------------
begin
comunicacion:com_usart
       generic map(sz_dat)
       port map(    Clk=>clk,
              reset=>reset,
              InRx=>RxIn,                      --al bloque principal
              OutTx=>TxOut,                    --al bloque principal
              load_Tx=>load_Tx,         --vienen del mux tenia load_Tx1
              Dat2Tx=>dat2_Tx,          --vienen del mux tenia dataout1
              TxBusy=>TxBusy1,
              Rx_listo=>Rx_listo1,
              Dat_Rec=>Dat_Rec1);

procesa:imag
       generic map(sz_indx,dim_ram,sz_k,sz_rad)
```

```
	port map(    clk=>clk,
		reset=>reset,
		enable=>En_ProcImg1,
		P_imag=>P_Imag,
		DatRRam_listo=>datR_rdy1,
		DatWRam_listo=>datw_rdy1,
		op_x=>Dat2Proc,
		addr_ready=>addr_ready1,
		ram_addr=>Ram_Addr1,
		pxl_proc=>pxl_proc1,
		area=>Area_proc,
		En_AreaRam=>En_AreaRam1,
		ReWe=>ReWe_proc,
		fin_proc=>proc_ready);          --al bloque principal

carg_env:load_send1
	generic map(sz_dat,dim_ram)
	port map(    Clk=>clk,
		Reset=>reset,
		enable=>En_LdSd11,
		clr_cnt=>clr_cnt,
		env_car=>Sel_EnvCar1,
		datin_Ld=>Dat_Rec1,
		datin_Sd=>Dat2Proc,
		datout=>datout1,
		area_in=>area_in1,
		re_we=>ReWe_LdSd,
		En_Mem=>EnMem_ldSd,
		address=>addr_LdSd,
		mem_fin=>Mem_Fin1,
		Tx_busy=>TxBusy1,
		load_Tx=>load_Tx1,
		Rx_ready=>Rx_listo1);

MemFin<=Mem_Fin1;

ctrl_todo:ctrl3b
	port map(clk=>clk,
		enable=>enable,           -- al bloque principal
		reset=>reset,
		proc_dat=>proc_dat,       -- al bloque principal
		fin_proc=>RecEnv,         -- al bloque principal,   RecEnv->1
		En_LdSd1=>En_LdSd11,
		Sel_EnvCar=>Sel_EnvCar1,
		Sel_ProcCom=>Sel_ProcCom1,
		lee_grab=>le_gr,
		En_ProcImg=>En_ProcImg1);

Ram_M:ram_1b
generic map(dim_ram)
port map(    clk=>clk,
	enable=>En_Mem1,
	re_we=>ReWe_Mem1,
	address=>Addr_Mem1,
	datain=>DatIn_Mem1,
	dataout=>Dat2Proc,
	datR_rdy=>datR_rdy1,
	datW_rdy=>datW_rdy1);

selectores:muxs
	generic map(dim_ram)
```

```
       port map(sel_PrCom=>Sel_ProcCom1,
              EnMem_Proc=>addr_ready1,
              Addr_Proc=>Ram_Addr1,
              DatOut_Proc=>pxl_proc1,
              ReWe_Proc=>ReWe_proc,
              EnMem_Com=>EnMem_LdSd,
              Addr_Com=>addr_LdSd,
              DatOut_Com=>datout1(0),
              ReWe_Com=>ReWe_LdSd,
              En_Mem=>En_Mem1,
              Addr_Mem=>Addr_Mem1,
              DatIn_Mem=>DatIn_Mem1,
              ReWe_Mem=>ReWe_Mem1);

area_res:area_block1
       generic map(dim_ram, 8)
       Port map(clk=>clk,
              reset=>reset,
              enable=>enable,
              Ar_In=>Area_proc,
              Ar_Op=>area_in1,
              ArOp_rdy=>En_AreaRam1,
              clr_cnt=>clr_cnt,
              lee_grab=>le_gr,
              Busy_Tx=>TxBusy1,
              Byte_Ar=>byte_area,
              Ld_Tx=>Ld_Tx);

area_mux: mux_area
       generic map (7)
       Port map(sel_SlAr=>le_gr,
              load_SdLd=>load_Tx1,
              dat_SdLd=>datout1,
              load_Ar=>Ld_Tx,
              dat_SdAr=>byte_area,
              load_tx=>load_Tx,
              dat2tx=>dat2_Tx
              );

led<=Area_proc(12);
--para ver como se comportan diferentes señales durante el procesamiento.
ve_dato<=Addr_Mem1(0);
ve_dato2<=Dat2Proc;
ve_dato3<=DatIn_Mem1;
ve_dato4<=datR_rdy1;
ve_dato5<=En_Mem1;
end BEHAVIORAL;
```

Listado bloque de comunicación

```
----------------------------------------------------
--bloque de comunicación, configura un puerto USART
----------------------------------------------------

library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity com_usart is
generic (sz_dat: integer:=8);
port (Clk : in std_logic;
       reset : in std_logic;
       InRx : in std_logic;
       OutTx : out std_logic;
       load_Tx : in std_logic;
       Dat2Tx : in  Std_Logic_Vector(7 downto 0);
       TxBusy : out Std_Logic;
       Rx_listo: out Std_Logic;                        -- Byte disponible
       Dat_Rec : out Std_Logic_Vector(7 downto 0)    -- Byte recibido
       );
end com_usart;

architecture BEHAVIORAL of com_usart is
--baud_rate
signal clk_16 : std_logic;

-------------------------------------------------
--componentes.
-------------------------------------------------
component baud_rate
       port (Clk : in std_logic;
             Reset : in std_logic;
             clk_16 : out std_logic);
end component;

component USART is
       port (Clk : in Std_Logic;
             Clk_16 : in  Std_Logic;
             Load : in Std_Logic;
             Reset : in Std_Logic;
             Rx : in Std_Logic;                        --bit de recepción
             Dat_Tx : in Std_Logic_Vector(7 downto 0);--Byte a transmitir
             Tx : out Std_Logic;                       -- bit de salida serial
             Tx_busy :out Std_Logic;
             Rx_ready : out Std_Logic;                 -- Byte disponible
             Reg_out: out Std_Logic_Vector(7 downto 0));   -- Byte recibido
end component;
-------------------------------------------------
 --Termina declaración de componentes
-------------------------------------------------
begin

rel_RxTx : baud_rate
       port map (Clk=>Clk,
             Reset=>reset,
             clk_16=>clk_16);

com_usart : USART
       port map(
             Clk=>Clk,
             Clk_16=>clk_16,
             load=>load_Tx,
             Reset=>reset,
             Rx=>InRx,
             Dat_Tx=>Dat2Tx,
             Tx=>OutTx,
             Tx_busy=>TxBusy,
             Rx_ready=>Rx_listo,
             Reg_out=>Dat_Rec
```

```
            );
end BEHAVIORAL;
```

Listado bloque de baud rate

```
--  ------------------------------------------------------------------------
--------
-- genera el Baud rate para el usart
-- genera el reloj Clk_16, duracion del pulso en alto - un ciclo de reloj
principal
--  ------------------------------------------------------------------------
---------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity baud_rate is
port (Clk : in Std_Logic;
      Reset : in Std_Logic;
      clk_16 : out Std_Logic);
end entity;

architecture Behaviour of baud_rate is
-- --------------------------
-- Clk16 Clock Generation
-- --------------------------

begin
process (Reset, Clk)
variable Div16 : integer range 0 to 324;
constant divisor: integer:=324;        --325 para un baud rate de 9600
                                       --(poner 324, el cero tambien cuenta)
                                       --valor que se desea 153600
                                       --valor obtenido     153846
begin
if Reset='1' then
      Clk_16 <= '0';
      Div16 := 0;
elsif rising_edge(Clk) then
      Clk_16 <= '0';
      if Div16 = Divisor then
            Div16 := 0;
            Clk_16 <= '1';
      else
            Div16 := Div16 + 1;
      end if;
end if;
end process;
end Behaviour;
```

Listado bloque USART.

```
-----------------------------------------------------
--bloque usart.
--configura el puerto seria del tarjeta nexys2
-----------------------------------------------------

library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity USART is
port (Clk : in Std_Logic;
      Clk_16 : in Std_Logic;
      Load : in Std_Logic;
      Reset : in Std_Logic;
      Rx : in Std_Logic;                              -- bit de recepcion
      Dat_Tx : in Std_Logic_Vector(7 downto 0);     -- Byte a transmitir
      Tx : out Std_Logic;                             -- bit de salida serial
      Tx_busy : out Std_Logic;
      Rx_ready : out Std_Logic;                       -- Byte disponible
      Reg_out : out Std_Logic_Vector(7 downto 0));        -- Byte recibido
end entity;

architecture Behaviour of USART is
type EstTx is (Idle, Load_Tx, Stop_Tx, Shift_Tx);
signal Tx_Edo : EstTx;
type EstRx is (Idle, Start_Rx, Stop_Rx, Shift_Rx, Edge_Rx, Rx_err);
signal Rx_Edo : EstRx;
signal clk_sync : Std_Logic;
signal clk_Tx : Std_Logic;
signal clk_Rx : Std_Logic;

begin
-- --------------------------
-- Tx Clk
-- reloj en alto - un ciclo de reloj principal
-- --------------------------
reloj_Tx: process (Reset, Clk)
variable Clk_Div : integer range 0 to 15;
variable aux1: bit;
begin

if Reset='1' then
      Clk_Div := 0;
      clk_Tx <= '0';
      aux1 :='0';
elsif rising_edge(Clk) then
      clk_Tx <= '0';
      if (Clk_16='1') then
            if Clk_Div = 15 then            --1 para smular
                  clk_Tx<='1';
                  Clk_Div := 0;
            else Clk_Div := Clk_Div + 1;
            end if;
      end if;
end if;
end process;

-- ------------------------------
-- Rx Clock reloj de muestreo
-- reloj en alto - un ciclo de reloj principal
-- ------------------------------
reloj_Rx:process (Reset, Clk)
variable RxDiv : integer range 0 to 7;

begin
```

```
if Reset='1' then
       clk_Rx <= '0';
       RxDiv := 0;
elsif rising_edge(Clk) then
       clk_Rx <= '0';
       if clk_sync='1' then
              RxDiv := 0;
       elsif Clk_16='1' then
              if RxDiv = 7 then
                     RxDiv := 0;
                     clk_Rx <= '1';
              else   RxDiv := RxDiv + 1;
              end if;
       end if;
end if;
end process;

-- -------------------------
-- Transmisor
-- -------------------------
Transmisor: process (Reset, Clk)
variable TxBusy : std_logic;
variable TxBitPos: integer range 0 to 8;
variable Reg_Ent : Std_Logic_Vector(7 downto 0);
variable Tx_reg : Std_Logic_Vector(8 downto 0);

begin
if Reset='1' then
       Tx_Reg := (others => '1');
       TxBitPos := 8;
       Tx_Edo <= idle;
       TxBusy := '0';
       Reg_Ent := (others => '0');
       Tx <= '1';
elsif rising_edge(Clk) then
       case Tx_Edo is
       when Idle =>
              if load='1' then
                     Reg_Ent := Dat_Tx;
                     TxBusy := '1';
                     Tx_Edo <= Load_Tx;
              else
                     TxBusy := '0';
              end if;
       when Load_Tx =>
              if clk_Tx='1' then
                     Tx_Edo <= Shift_Tx;
                     TxBitPos := 0;            --bits a transmitir
                     Tx_reg := Reg_Ent & '0';
              end if;
       when Shift_Tx =>
              if clk_Tx='1' then
                     Tx <= Tx_reg(TxBitPos);
                     if TxBitPos=8 then
                            Tx_Edo <= Stop_Tx;
                     end if;
                     TxBitPos := TxBitPos + 1;
              end if;
       when Stop_Tx =>
              if clk_Tx='1' then
                     Tx_Edo <= Idle;
```

```
                Tx <= '1';
            end if;
      when others =>
            Tx_Edo <= Idle;
      end case;
end if;
Tx_busy<=TxBusy;
end process;

-- ------------------------
-- Receptor
-- ------------------------
Receptor: process (Reset, Clk)
variable RxBitPos : integer range 0 to 8;
variable Rx_Reg: std_logic_vector(7 downto 0);
variable R_listo:std_logic;
constant Nobits:integer:=8;

begin

if Reset='1' then
      Rx_Reg := (others => '0');
      Reg_out <= (others => '0');
      RxBitPos := 0;
      Rx_Edo <= Idle;
      R_listo := '0'; --dato listo
      clk_sync <= '0';
elsif rising_edge(Clk) then
      clk_sync <= '0';

      if R_listo='1' then -- con el if clk_16 ->
            R_listo:='0';--dato listo permanece en 1 un pulso declk_16
      end if;

      case Rx_Edo is
      when Idle =>              -- espera bit de inicio
            RxBitPos := 0;
            if Rx='0' then
                  Rx_Edo <= Start_Rx;
                  clk_sync <='1';
            end if;
      when Start_Rx =>                -- espera el primer bit del dato
            if clk_Rx = '1' then
                  if Rx='1' then      -- error deberia seguir siendo cero.
                        Rx_Edo <= Rx_err;
                  else
                        Rx_Edo <= Edge_Rx;
                  end if;
            end if;
      when Edge_Rx =>
            if clk_Rx = '1' then
                  if RxBitPos = Nobits then
                        Rx_Edo <= Stop_Rx;
                  else
                        Rx_Edo <= Shift_Rx;
                  end if;
            end if;
      when Shift_Rx =>    --se encuentra aprox. en medio del pulso
                          --[edge, shift, edge]
            if clk_Rx = '1' then
                  Rx_Reg(RxBitPos) := Rx;
```

```
                  RxBitPos := RxBitPos + 1;
                  Rx_Edo <= Edge_Rx;
            end if;
      when Stop_Rx =>
            if clk_Rx = '1' then
                  Reg_out <= Rx_reg;
                  R_listo :='1';
                  Rx_Edo <= Idle;
            end if;
      when Rx_err => -- Error
            if Rx='1' then
                  Rx_Edo <= Idle;
            end if;
      end case;
end if;
Rx_ready<=R_listo;
end process;
end Behaviour;
```

Listado multiplexores.

Se diseñaron dos multiplexores de 2 a 1 de diferente tamaño de bus, uno para las direcciones de la memoria RAM, y el otro para los datos. Como se esta trabajando con imágenes binarias, los datos son de 1 bit.

Listado multiplexor 2 a 1 de genérico.

```
--------------------------------------------------
--multiplexor 2 a 1 para las direcciones de memoria
--------------------------------------------------
------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
------------------------------------------------

Entity  Mux_2a1 is
Generic(dim_mem: integer:=14);
Port (Sel : in std_logic;                          --selector
      x : in std_logic_vector(dim_mem downto 0);
      y : in std_logic_vector(dim_mem downto 0);
      s : out std_logic_vector(dim_mem downto 0)    --salida
      );
End Entity Mux_2a1;

Architecture Multiplexer of Mux_2a1 is
Begin
s <=   x when sel= '0' else      --si 'sel', es cero, entonces la señal
                                 --de salida 's' es la señal de entrada 'x'
       y when sel = '1' else     --si 'sel', es uno, entonces la señal
                                 --de salida 's' es 'y'.
       others => 'Z');           --Cuando no se cumple ninguna condicion
                                 --entonces la señal 's' se indefine
End Architecture Multiplexer;
```

Listado multiplexor 2 a 1 de 1 bit

```
------------------------------------------------
-- multiplexor 2 a 1 de un bit
------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
------------------------------------------------

Entity  Mux_2a1_1b is
Port   (sel: in std_logic;                    --selector
       x : in std_logic;                      --señal a elegir
       y : in std_logic;
       s : out std_logic                      --salida
       );
End Entity Mux_2a1_1b;

Architecture Mux_1b of Mux_2a1_1b is
Begin
s <=   x when sel= '0' else      --si 'sel', es cero, entonces la señal.
                                 --de salida 's' es la señal de entrada 'y'.
       y when sel = '1' else     --si 'sel', es uno, entonces la señal
                                 -- de salida 's' es 'y'.
       'Z';                      --Cuando no se cumple ninguna condicion
                                 --entonces la señal 's' se indefine.
End Architecture Mux_1b;
```

Listado bloque de memoria.

```
-----------------------------------------------------
--ram de 1 bit x (2 x No. pixeles de la imagen)
-----------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ram_1b is
       generic (dim_ram : integer :=14);      --14 imagen de 128 x 128 x 2
                                              --13 imagen de 64 x 64 x 2
       port (clk : in  std_logic;
             enable: in  std_logic;
             re_we : in  std_logic;
             address : in  std_logic_vector(dim_ram downto 0);
             datain : in  std_logic;
             dataout : out std_logic;
             datR_rdy : out std_logic;
             datW_rdy : out std_logic
             );
end entity ram_1b;

architecture RTL of ram_1b is

type ram_type is array ((2**address'length)-1 downto 0) of std_logic;
```

```
signal ram : ram_type;
signal datR_listo : std_logic;
signal datW_listo : std_logic;

begin
Ram_read: process(clk,enable,address,re_we)
begin
if rising_edge(clk) then
       if (enable='1' and re_we = '1') then
              dataout <= ram(conv_integer(address));
              datR_listo <= '1';
       else   dataout <= '0';
              datR_listo <= '0';
       end if;
       datR_rdy <= datR_listo;
end if;
end process Ram_read;

Ram_write: process(clk,address,enable,datain,re_we)
begin
if rising_edge(clk) then
       if (enable='1' and re_we = '0') then
              ram(conv_integer(address)) <= datain;
               datW_listo <= '1';
       else datW_listo <= '0';
       end if;
       datW_rdy <= datW_listo;
end if;
end process Ram_write;

end RTL;
```

Listado bloque Load_Send1

```
--------------------------------------------------------------------------------
------
--utiliza el usart para enviar la imagen procesada a la pc
--la salida de datos para guardar en la memoria es dataout(0)
--------------------------------------------------------------------------------
------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity load_send1 is
generic (sz_dat : integer := 8;
       dim_mem : integer := 14);        -- 13 bits para 4096 -> 64x64
                                        -- 14 bits para 16383 -> 128x128
port (Clk : in Std_Logic;
       Reset : in Std_Logic;
       enable : in Std_Logic;
       clr_cnt : in std_logic;
       env_car : in std_logic;
       datin_Ld : in std_logic_vector(sz_dat-1 downto 0);   --del usart
       datin_Sd : in std_logic;                --de la memoria de salida
       datout : out std_logic_vector(sz_dat-1 downto 0);
       area_in : out std_logic_vector(dim_mem-1 downto 0);
```

```
--memoria
       re_we  : out std_logic;
       En_Mem : out std_logic;
       address: out std_logic_vector(dim_mem downto 0);
       mem_fin : out std_logic;
--para enviar
       Tx_busy : in Std_Logic;
       load_Tx : out Std_Logic;
--para recibir
       Rx_ready : in Std_Logic                        -- Byte disponible.
       );
end entity;

architecture Behaviour of load_send1 is

type Est_TxRx is (Rx_listo, Load_Addr, pausa, D_listo, W_TxBusy, Next_Addr,
Stand_By);
signal Edo_TxRx : Est_TxRx;
signal M_full : std_logic;
constant final : integer := 16384;            --4096=>64x64  16384=>128x128

begin
process (clk,reset)
variable cnt : natural range 0 to 16384;      --4096=>64x64  16384=>128x128
variable area_1 : integer range 0 to 16383;
variable aux_addr: std_logic_vector(dim_mem-1 downto 0);
variable dat_1b : std_logic;
variable aux : bit;
variable aux_2 : integer range 0 to 4;

begin
if Reset = '1' then
       address <= (others=>'0');
       En_Mem <= '0';
       re_we <= '1';
       load_Tx <= '0';
       Edo_TxRx <= Stand_by;                   --estado inicial en espera.
       M_full <= '0';
       mem_fin <= '0';
       area_1 := 0;
       cnt := 0;
       aux := '0';
       datout <= (others => '0');
       area_in <= (others => '0');
       aux_2 := 0;
elsif rising_edge(clk) then
       if enable = '1' then
              case Edo_TxRx is
              when Rx_listo =>
                     if Rx_ready = '1' then
                            if conv_integer(datin_Ld) >= 1 then    --umbral
                                   dat_1b := '1';
                                   area_1 := area_1 + 1;
                            else   dat_1b := '0';
                            end if;
                            Edo_TxRx <= Load_Addr;
                     end if;
              when Load_Addr =>
                     if env_car = '1' then
                            re_we <= '1';  -- habilita lectura de memoria RAM
                     else   datout <= (0 => dat_1b, others => '0');
```

```
                re_we <= '0';-- habilita grabación de memoria RAM
            end if;
            aux_addr := conv_std_logic_vector(cnt,address'length-1);
            address <= '0' & aux_addr;
            Edo_TxRx <= pausa;
      when pausa =>
            En_Mem <= '1';
            if aux_2 = 2 then
                  if env_car = '1' then
                        Edo_TxRx <= D_listo;
                  else  Edo_TxRx <= Next_Addr;
                  end if;
                  aux_2 := 0;
            else aux_2 := aux_2 + 1;
            end if;
      when D_listo =>
            if datin_Sd = '1' then
                  datout <= (others => '1');--manda "11111111"
            else  datout <= (others => '0');--manda "00000000"
            end if;
            Edo_TxRx <= W_TxBusy;
      when W_TxBusy =>
            if (Tx_busy = '0') then
                  load_Tx <= '1';
                  Edo_TxRx <= Next_Addr;
            end if;
      when Next_Addr =>
            En_Mem <= '0';
            load_Tx <= '0';
            cnt := cnt + 1;
            if cnt = final then
                  aux := '1';
                  Edo_TxRx <= Stand_By;
            else
                  if env_car = '1' then
                        Edo_TxRx <= Load_Addr;
                  else  Edo_TxRx <= Rx_listo;
                  end if;
            end if;
      when Stand_by =>
            En_Mem <= '0';
            if aux = '1' then
                  M_full <= '1';
           area_in <= conv_std_logic_vector(area_1, area_in'length);
            end if;
            if clr_cnt = '1' then
                  cnt := 0;
                  M_full <= '0';
                  if env_car = '1' then
                        Edo_TxRx <= Load_Addr;
                  else  Edo_TxRx <= Rx_listo;
                  end if;
                  aux := '0';
            end if;
      when others => null;
      end case;
else  re_we <= '1';
      M_full <= '0';
      aux := '0';
end if;
mem_fin <= M_full;
```

```
end if;
end process;
end Behaviour;
```

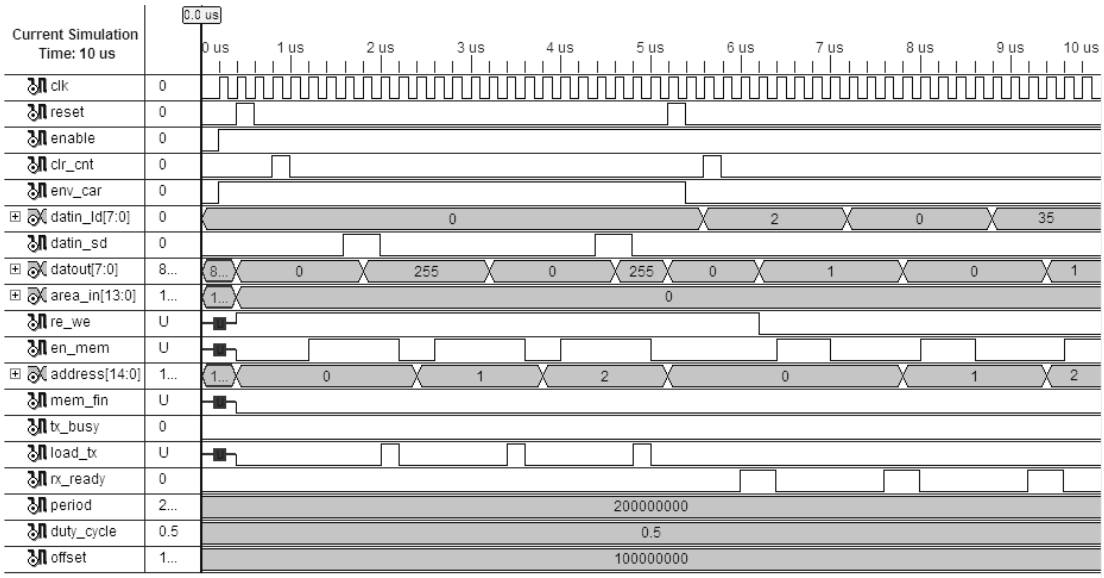

Figura A. 1. Simulación bloque load_send

La simulación muestra el funcionamiento del bloque. La señal enable en 1 lo habilita, y reset en 1 lo pone en condiciones iniciales.

De 0 a 5us se configura para enviar los datos guardados en la memoria RAM a la PC, con la entrada env_car =1. Se configura la memoria RAM para leer (señal re_we en 1), se generan las direcciones de la memoria en la salida “address”, un ciclo de reloj después habilita la memoria con la señal en_mem, y espera 3 ciclos de reloj para que la entrada datin_sd tenga el dato, si el dato es “1” el bloque envia “11111111” (255), si es “0” envia “00000000” por la salida datout. En el ejemplo se tienen tres bits para enviar “1” a los 1.6us, “0” a los 3us, “1” a los 4.4us. Finalmente el carga los datos en el bloque del usart utilizando la señal load_tx.

De 5us a 10us el bloque se configura para recibir. Se configura la memoria para grabar (re_we = 0), se reinicializa el contador de direcciones (se queda en la 0), se usa datout(0) para enviar el bit a la entrada de datos de la memoria. Cada vez que la señal rx_ready va a 1 se lee la entrada datin_ld, si

es mayor o igual a 1 datout(0) es 1, si es cero datout(0) es 0, un ciclo de reloj despues de que datout(0), se habilita la memoria.

Listado bloque ctrl3b

```
----------------------------------------------------
--controla la operación del circuito
--por medio de dos interruptores
--para cargar, procesar, enviar la imagen
--o para enviar el vector de resultados
----------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ctrl3b is
port (clk : in std_logic;
       enable : in std_logic;
       reset : in  std_logic;
       proc_dat : in std_logic;
       fin_proc : in std_logic;
              En_LdSd1 : out std_logic;
              Sel_EnvCar : out std_logic;
              Sel_ProcCom : out std_logic;
              lee_grab :out std_logic;
              En_ProcImg : out std_logic);
end entity ctrl3b;

architecture behavior of ctrl3b is
begin

process (clk,reset)
begin

if Reset = '1' then
       En_ProcImg <= '0';
       Sel_EnvCar <= '0';              --load_send1 modo-carga
       Sel_ProcCom <= '0';             --el control lo tiene load_send1
       lee_grab <= '0';
       En_LdSd1 <= '0';
elsif rising_edge(clk) then
       if enable = '1' then
              if (proc_dat = '0' and fin_proc = '0') then   --carga imagen--
                     Sel_ProcCom <= '0';       --mux's - control a usart_com
                     Sel_EnvCar <= '0';        --load_send1 modo-carga
                     En_ProcImg <= '0';  --deshabilita procesado de la imagen
                     En_LdSd1 <= '1';          --habilita load_send1
                     lee_grab <= '0';--deshabilita send_area, area_pecstrum
                                    --modo resta, ctrl_area1 modo graba
                                    --mux's control a load_send1
              end if;

              if (proc_dat = '1' and fin_proc = '0') then   --procesa imagen
                     En_LdSd1 <= '0';          --deshabilta load_send1
                     Sel_ProcCom <= '1';       --mux's - control a imag
                     Sel_EnvCar <= '1';        --load_send1 modo-envia
                                               --bloque esta off
```

```
                En_ProcImg <= '1';
                lee_grab <= '0';
            end if;

            if (proc_dat = '0' and fin_proc = '1') then   --envia imagen--
                En_ProcImg <= '0';
                Sel_ProcCom <= '0';       --mux's -control a usart_com
                Sel_EnvCar <= '1';        --load_send1 modo-envia
                En_LdSd1 <= '1';          --habilta load_send1
                lee_grab <= '0';
            end if;

            if (proc_dat = '1' and fin_proc = '1') then   --envia vector--
                En_LdSd1 <= '0';          --deshabilta load_send1
                Sel_EnvCar <= '1';  --load_send1 modo-envia bloque off
                Sel_ProcCom <= '0'; --mux's - control a usart_com
                En_ProcImg <= '0';  --deshabilita procesado de imagen
                lee_grab <= '1';    --habilita send_area, area_pecstrum
                                    --modo lee, ctrl_area1 modo lee
                                    --mux's control a area_block
            end if;
        end if;
end if;
end process;
end behavior;
```

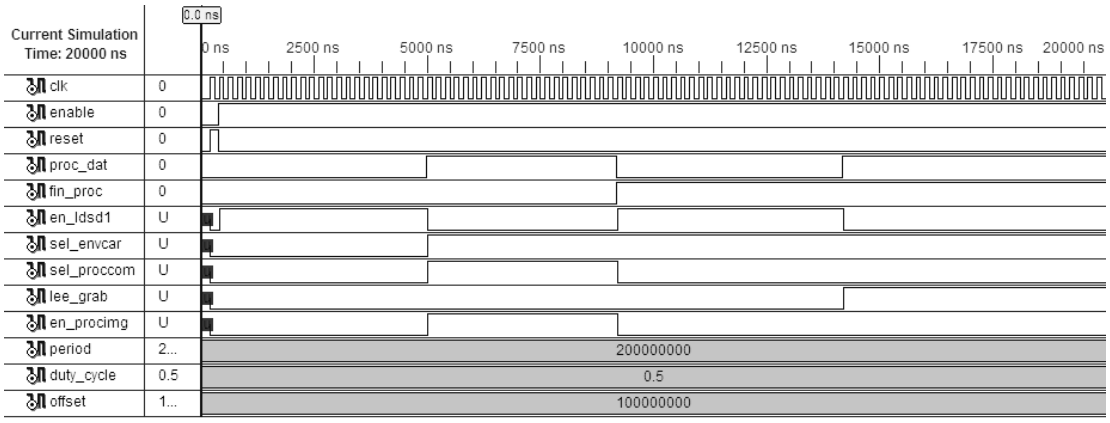

Figura A. 2. Simulación bloque ctrl3b

La simulación muestra muestra el funcionamiento del bloque ctrl3b, las entradas fin_proc y proc_dat van conectadas a dos interruptores (sw7 y sw6 respectivamente) de la tarjeta nexys2, se utilizan para seleccionar el modo de operación del programa.

Con (0,0) se configura para cargar la imagen, se habilita el bloque load_send1 con la señal En_LdSd1 = 1 y se configura en modo receptor con Sel_EnvCar=0, los mux's se configuran para comunicación con

Sel_ProcCom=0, el bloque de areas en modo resta con lee_grab=0 y el bloque de procesamiento se deshabilita.

Con (0,1) se configura para procesar la imagen, se deshabilita load_send1 (En_LdSd=0), los mux's se configuran para procesamiento (Sel_ProcCom=1) y se habilita el bloque de procesamiento (En_ProcImg=1).

Con (1,0) se configura para enviar la imagen, se habilita load_send (En_LdSd=1) en modo transmisor (Sel_EnvCar=1), los mux's se configuran para comunicación (Sel_ProcCom=0).

Finalmente con (1,1) se configura para enviar el vector de resultados, todos los bloques se deshabilitan menos el bloque de restas, que se configura para enviar el vector (lee_grab=1).

Bloques que forman al bloque area_block

Listado bloque area_pecstrum

```
----------------------------------------------------------------------------
------
--area_pecstrum hace las restas de las areas resultantes entre aperturas
--y guarda el resultado en una memoria ram
--indx se incrementa cuando enable va a 1
----------------------------------------------------------------------------
------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity area_pecstrum is
generic (dim_addr : integer := 5;
       dim_ram : integer := 14);
Port ( clk : in STD_LOGIC;
       enable : in STD_LOGIC;
       reset : in STD_LOGIC;
       lee_grab : in STD_LOGIC;
       clr_indx : in STD_LOGIC;
       Area_in : in STD_LOGIC_VECTOR (dim_ram-1 downto 0);
       Area_op : in STD_LOGIC_VECTOR (dim_ram-1 downto 0);
       Area_aper : out STD_LOGIC_VECTOR (dim_ram-1 downto 0);
       area_rdy : out STD_LOGIC;
       last_ar : out STD_LOGIC
       );
end area_pecstrum;

architecture Behavioral of area_pecstrum is
```

```
type    ram_type    is    array    ((2**dim_addr)-1    downto    0)    of
std_logic_vector(dim_ram-1 downto 0);
signal ram : ram_type;

begin

process(clk,reset,enable)
variable indx : integer range 0 to 31;              --numero de aperturas
variable indx_2 : integer range 0 to 31;
variable var_0 : std_logic_vector(dim_ram-1 downto 0);
variable var_1 : std_logic_vector(dim_ram-1 downto 0);
variable var_2 : std_logic_vector(dim_ram-1 downto 0);
variable in_dx : bit;
variable A_rea : bit;
begin

if (reset='1') then
      Area_aper <= (others=>'0');
      indx := 0;
      indx_2 := 31;
      A_rea := '0';
      area_rdy <= '0';
      last_ar <= '0';
      in_dx:='0';
elsif (rising_edge(clk)) then
      if clr_indx = '1' then
            indx := 0;
      end if;
      if enable = '1' then
            case A_rea is
            when '0' =>
                  if lee_grab = '0' then          -- grabacion de ram
                        var_1 := Area_in;
                        var_0 := Area_op;
                        case in_dx is
                        when '0' =>  -- resta de areas
                                    -- graba los resultados en memoria
                              ram(indx) <= var_0 - var_1;
                              var_2 := var_1;
                              in_dx := '1';
                        when '1' =>
                              ram(indx) <= var_2 - var_1;
                              var_2 := var_1;
                        end case;
                              indx_2 := indx;
                  else                        --lectura de ram
                        if indx <= indx_2 then
                            Area_aper <= ram(indx);--resultado de restas
                        else last_ar <= '1';
                        end if;
                  end if;
                  area_rdy <= '0';
                  A_rea := '1';
                  indx := indx+1;
            when '1' =>
                  area_rdy <= '1';
            end case;
      else  A_rea := '0';
            area_rdy <= '0';
      end if;
end if;
```

```
end process;
end Behavioral;
```

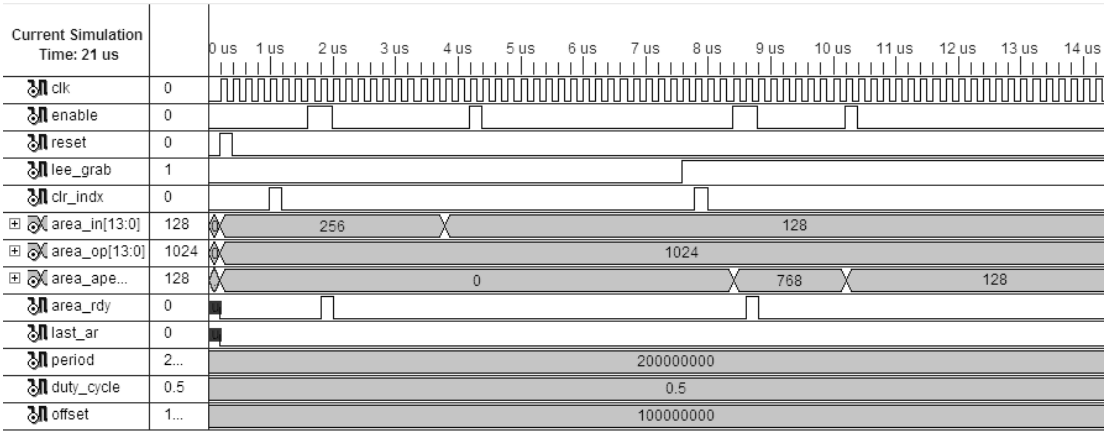

Figura A. 3. Simulación bloque area_pecstrum.

En la figura A3 se tiene el resultado del bloque area_pecstrum, el dato que se encuentra en la entrada area_op, es el area de la imagen original, la señal area_in es el area de la imagen después de una apertura. El modo de operación se seleciona con la entrada lee_grab; 0 para modo resta y 1 para modo lee. Cada vez que enable va a 1 el programa realiza una resta, o saca un resultado (area_aper). En modo resta este bloque realiza la resta de las areas, la primera resta la hace con el area de la imagen original (area_op) y con el area resultante después de la primera apertura (area_in). En el ejemplo enable va a 1 en 1.6us y los valores que se tienen son 1024 (area_op) y 256 (area_in). Cuando enable va a1 nuevamente (4.2us) la resta se efectua con el area anterior y el area nueva (128.). Posteriormente el bloque se configura en modo lee, y con cada pulso de enable muestra un resultado, en la simulación se puede observar los resultados de las dos primeras restas 1024-256=768 y 256-128=128.

Listado bloque ctrl_area1.

```
----------------------------------------------------------------------------
------
--bloque ctrl_area1
--controla las señales para hacer las restas y guardar los resultados en
memoria
--configura las señales para enviar el vector de resultados a la PC
----------------------------------------------------------------------------
------
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ctrl_area1 is
Port ( clk : in  STD_LOGIC;
       enable : in  STD_LOGIC;
       reset : in  STD_LOGIC;
       send_rest : in  STD_LOGIC;       --de ctrl3b (send_rest)
       clr_cnt : in  STD_LOGIC;
       envia2B : in  STD_LOGIC;
       last_2B : in  STD_LOGIC;
       ArOp_rdy : in  STD_LOGIC;
       ArApecs_rdy : in  STD_LOGIC;
       ini_SdArea : out  STD_LOGIC;     --para enviar la palabra (16 bits)
       En_Apecs : out STD_LOGIC;
       clr_indx : out STD_LOGIC);
end ctrl_area1;

architecture Behavioral of ctrl_area1 is
type EstSdArea is (A, B, C, D, F);
signal Sd_Edo : EstSdArea;
type EstPrcArea is (inicio, pausa, fin);
signal Proc_Edo : EstPrcArea;
begin
process(clk,enable,reset)
variable aux : bit;
begin
if (reset='1') then
       ini_SdArea <= '0';
       En_Apecs <= '0';
       clr_indx <= '0';
       Proc_Edo <= inicio;
       Sd_Edo <= A;
elsif (rising_edge(clk)) then
       if (enable)='1' then
              if send_rest = '0' then                        ----de ctrl3a----
                     ini_SdArea <= '0';
                     case Proc_Edo is
                     when inicio =>
                            if ArOp_rdy = '1' then
                                   En_Apecs <= '1';    --area_pecstrum se
                                   -- incrementa en 1 cuando enable va a 1
                                   Proc_Edo <= pausa;
                            else En_Apecs <= '0';
                                   Proc_Edo <= inicio;
                            end if;
                     when pausa =>
                            if ArApecs_rdy = '1' then
                                   Proc_Edo <= fin;
                            else Proc_Edo <= pausa;
                            end if;
                     when fin =>
                            En_Apecs <= '0';
                            Proc_Edo <= inicio;
                     end case;
              else
                     case Sd_Edo is
                     when A =>
                            if clr_cnt = '1' then
```

```
                        clr_indx <= '1';    --reinicia contador del
                                            --vector
                        Sd_Edo <= B;
                  else Sd_Edo <= A;
                  end if;
            when B =>
                  Sd_Edo <= C;
            when C =>
                  clr_indx <= '0';          --dura un pulso de reloj
                  Sd_Edo <= D;
            when D =>
                  En_Apecs <= '1';
                  if ArApecs_rdy = '1' then
                        ini_SdArea <= '1';  --send_area envia
                                    --2 bytes (resultado resta area)
                        Sd_Edo <= F;
                  else ini_SdArea <= '0';
                        Sd_Edo <= D;
                  end if;
            when F =>
                  ini_SdArea <= '0';
                  En_Apecs <= '0';
                  if envia2B = '1' then
                        if last_2B = '1' then
                              Sd_Edo <= A;
                        else Sd_Edo <= D;
                        end if;
                  else Sd_Edo <= F;
                  end if;
            end case;
      end if;
   end if;
end if;
end process;
end Behavioral;
```

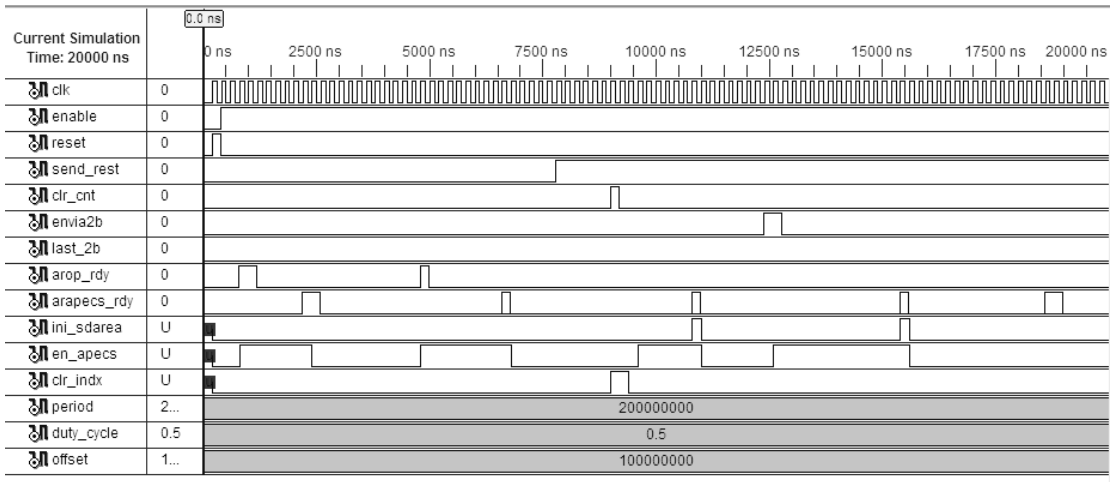

Figura A. 4. Simulación bloque ctrl_area1.

Listado bloque send_area.

```
----------------------------------------------------------------------------
------
--send_area envia los resultados de las restas en dos bytes
--el primer byte contiene la parte alta de resultado (bits 14 al 8)
--y el segundo la parte baja (bits 7 al 0)
----------------------------------------------------------------------------
------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity send_area is
generic (dim_ram : integer := 14;
      byte : integer := 8);
Port ( clk : in  STD_LOGIC;
      enable : in  STD_LOGIC;
      reset : in  STD_LOGIC;
      busy_tx : in  STD_LOGIC;
      inicio : in  STD_LOGIC;
      Aper_area : in  STD_LOGIC_VECTOR (dim_ram-1 downto 0);
      byte_ar : out  STD_LOGIC_VECTOR (byte-1 downto 0);
      load_tx : out STD_LOGIC;
      enviado : out STD_LOGIC
      );
end send_area;

architecture Behavioral of send_area is
type EstSd is (A, B, C, D, F);
signal Sd_Edo : EstSd;

begin
process(clk,enable,reset,busy_tx)
variable aux : bit;
begin
if (reset='1') then
      byte_ar <= (others=>'0');
      load_tx <= '0';
      aux := '0';
      Sd_Edo <= A;
      enviado <= '0';
elsif (rising_edge(clk)) then
      if (enable)='1' then
            case Sd_Edo is
            when A =>
                  if inicio = '1' then
                        Sd_Edo <= B;
                  else Sd_Edo <= A;
                  end if;
            when B =>
                  case aux is
                  when '0' =>
                        byte_ar <= "00" & Aper_area(dim_ram - 1 downto 8);
                        aux := '1';
                  when '1' =>
                        byte_ar <= Aper_area(7 downto 0);
                        aux := '0';
                  end case;
                  Sd_Edo <= C;
            when C =>
                  if busy_tx = '0' then
                        load_tx <= '1';
```

```
                        Sd_Edo <= D;
                  else load_tx <= '0';
                           Sd_Edo <= C;
                  end if;
            when D =>
                  if aux = '0' then
                        Sd_Edo <= F;
                        enviado <= '1';
                  else Sd_Edo <= B;
                  end if;
                  load_tx <= '0';
            when F =>
                  enviado <= '0';     --enviado dura un pulso de reloj
                  Sd_Edo <= A;
            end case;
      end if;
end if;
end process;
end Behavioral;
```

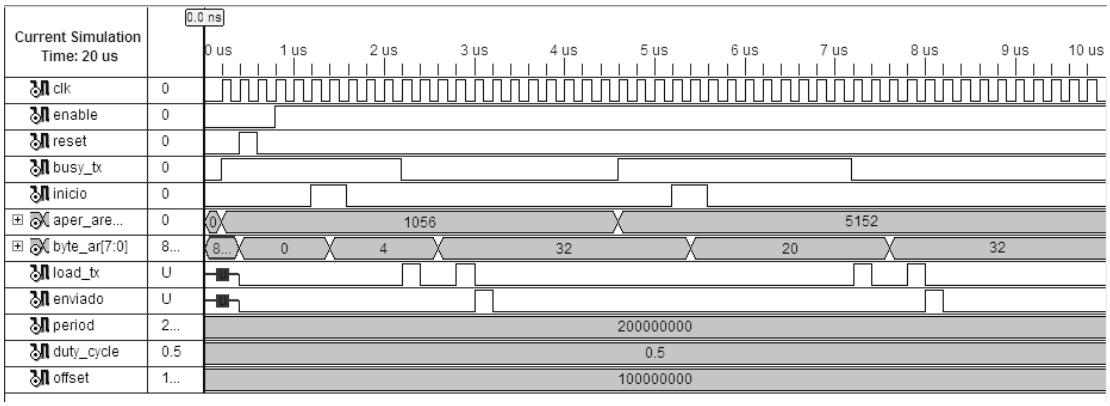

Figura A. 5. Simulación del bloque send_area.

Bloques que forman el bloque de procesamiento.

Listado bloque cont_pixel.

```
----------------------------------------------------------------------------
------
--genera las coordenadas del pixel (A, B) de la imagen. (A,B) inicia desde
0,0.
--se mueve sobre la matriz de la imagen
--la señal de dato_listo dura lo mismo que la señal de next_pxl
--la salida no cambia hasta el siguiente pulso next_pxl
----------------------------------------------------------------------------
------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity cont_pixel is
```

```
generic (sz_indx: integer:=8);   --sz_dat define el tamaño de los indx A,B
Port (clk : in STD_LOGIC;
      Reset : in STD_LOGIC;
      Enable : in STD_LOGIC;
      N : in  STD_LOGIC_VECTOR (7 downto 0);
      M : in  STD_LOGIC_VECTOR (7 downto 0);
      Next_Pxl : in  STD_LOGIC;
      A : out STD_LOGIC_VECTOR (sz_indx-1 downto 0);
      B : out STD_LOGIC_VECTOR (sz_indx-1 downto 0);
      last_pxl : out  STD_LOGIC;
      addr_listo : out std_logic);
end cont_pixel;
----------------------------------------------------------------------------
architecture Behavioral of cont_pixel is
signal ultimo_pxl: std_logic;

begin
process(clk,reset,enable,next_pxl)
variable cnt_A : integer range 0 to 255:=0;   --255-imagen de 256x256
variable cnt_B : integer range 0 to 255:=0;   --255-M y N maximos 256
variable aux : bit:='0';

begin
if reset = '1' then
      A <=(others => '0');
      B <=(others => '0');
      last_pxl <= '0';
      ultimo_pxl <= '0';
      cnt_A := 0;               --cambiar para iniciar desde otro pixel
      cnt_B := 0;               --cambiar para iniciar desde otro pixel
      aux := '0';
      addr_listo <= '0';
elsif (rising_edge(clk)) then
      if enable = '1' then
            if (Next_Pxl = '1') then
                  case aux is
                  when '0' =>
                        A <= conv_std_logic_vector(cnt_A,A'length);
                        B <= conv_std_logic_vector(cnt_B,B'length);
                        if (cnt_B+1 = conv_integer(M)) then
                              if (cnt_A+1= conv_integer(N)) then
                                    ultimo_pxl <= '1';
                              else cnt_A := cnt_A+1;
                                    cnt_B := 0;
                              end if;
                        else cnt_B := cnt_B+1;
                        end if;
                        addr_listo <= '0';
                        aux := '1';  --se ejecuta una vez
                  when '1' =>
                        addr_listo <= '1';
                  end case;
            else
                  aux := '0';
                  addr_listo <= '0';
            end if;
      else cnt_A := 0;                --reinicia contadores e indicadores
                                      --para procesar otra matriz
            cnt_B := 0;
            addr_listo <= '0';
            ultimo_pxl <= '0';
```

```
            aux := '0';
      end if;
      last_pxl <= ultimo_pxl;
end if;
end process;
end Behavioral;
```

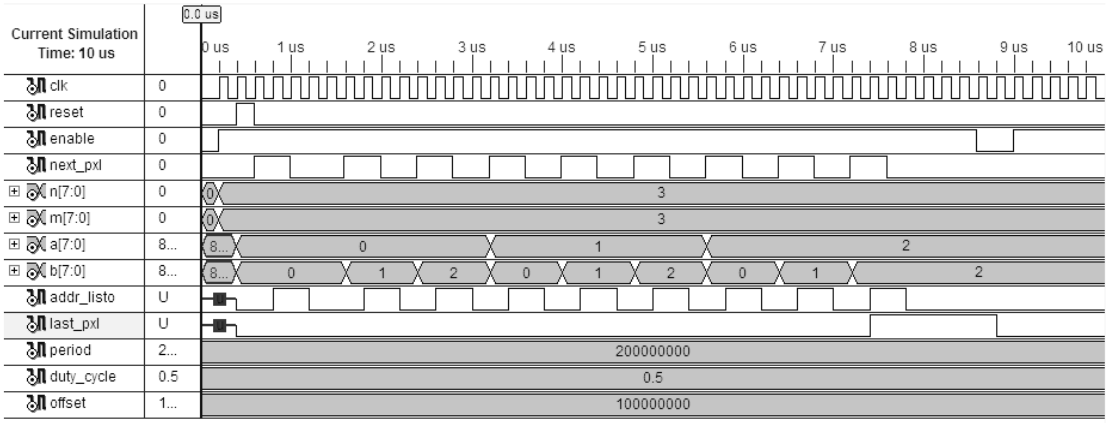

Figura A. 6. Simulación bloque cont_pixel

Listado bloque ctrl_SubMat1

```
----------------------------------------------------------------------------
------
--genera los subindices de (F,C) de la submatriz van desde el (0,0).
--la señal de dato_listo dura lo mismo que la de sig_k
--la salida no cambia hasta el siguiente pulso de sig_k
--sub_matF permanece en alto hasta que enable se haga cero.
--el tamaño del kernel lo controla -Rad- radio del kernel
----------------------------------------------------------------------------
------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ctrl_SubMat1 is
generic (size_k: integer:=7;
      sz_rad: integer:=6);                    --para un kernel de 64 x 64
Port ( clk : in STD_LOGIC;
      Reset : in STD_LOGIC;
      Enable : in STD_LOGIC;
      Rad : in STD_LOGIC_VECTOR (sz_rad-1 downto 0);
      sig_k : in STD_LOGIC;
      F : out STD_LOGIC_VECTOR (size_k-1 downto 0);
      C : out STD_LOGIC_VECTOR (size_k-1 downto 0);
      sub_matF : out STD_LOGIC;
      indx_listo : out std_logic);
end ctrl_SubMat1;
----------------------------------------------------------------------------
architecture Behavioral of ctrl_SubMat1 is
signal mat_f : std_logic;
```

```
begin
process(clk,reset,enable,sig_k)
variable cnt_F : integer range 0 to 65;
variable cnt_C : integer range 0 to 65;
variable K : integer range 0 to 65;
variable radio : integer range 0 to 32;
variable Aux_F : STD_LOGIC_VECTOR (F'range);
variable Aux_C : STD_LOGIC_VECTOR (C'range);
variable Aux : bit;

begin
if reset = '1' then
	F <=(others => '0');
	C <=(others => '0');
	sub_matF <= '0';
	mat_f <= '0';
	cnt_F := 0;
	cnt_C := 0;
	indx_listo <= '0';
	Aux := '0';
elsif (rising_edge(clk)) then
	if enable = '1' then
		if sig_k = '1' then
			case Aux is
			when '0' =>
				F <= conv_std_logic_vector(cnt_F,F'length);
				C <= conv_std_logic_vector(cnt_C,C'length);
				radio := conv_integer('0' & Rad);
				K := radio+radio+1;
				if (cnt_C+1 = K) then     --cnt_C+1 porque inicia
                                                  --en 0
					if (cnt_F+1 = K) then     --cnt_F+1 porque
                                                  --inicia en 0
						mat_f <= '1';
					else cnt_F := cnt_F+1;
						cnt_C := 0;
					end if;
				else cnt_C := cnt_C+1;
				end if;
				indx_listo <= '0';
				Aux := '1';
			when '1' =>
				indx_listo <= '1';
			end case;
		else
			Aux := '0';
			indx_listo <= '0';
		end if;
	else cnt_F := 0;                --reinicia contadores e indicadores
                                        --para procesar otra matriz
		cnt_C := 0;
		indx_listo <= '0';
		mat_f <= '0';
		Aux := '0';
	end if;
	sub_matF <= mat_f;
end if;
end process;
end Behavioral;
```

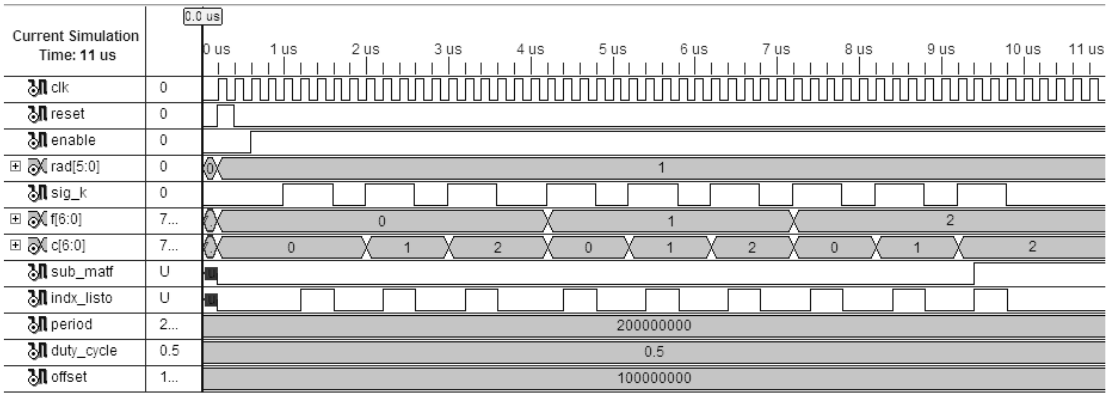

Figura A. 7. Simulación bloque ctrl_Sub_Mat1.

Listado bloque sub_mat.

```
----------------------------------------------------------------------------
------
--mapea la submatriz a la imagen, pone al pixel que se procesa en el centro
de la
--submatriz, los valores de la submatriz que queden fuera de la imagen se
llenan con
--el valor del pixel mas cercano.
--salida_lista dura lo mismo que enable
--el programa solo se ejecuta una vez con enable en alto
--la salida permanece sin cambio con enable = 0
----------------------------------------------------------------------------
------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sub_mat is
generic (sz_indx: integer:=8;
       size_k: integer:=7;             --para que F,C lleguen hasta 64 o 65
       sz_rad: integer:=6
       );
Port ( clk : IN std_logic;
       reset :IN std_logic;
       enable:IN std_logic;
       M : IN  std_logic_vector(7 downto 0);
       N : IN  std_logic_vector(7 downto 0);
       A : IN  std_logic_vector(sz_indx-1 downto 0); --A,B indices del píxel
                                                     --que se va a procesar
       B : IN  std_logic_vector(sz_indx-1 downto 0); --con 8 bits se
                                      -- direcciona 256 x 256 incluyendo 0
       F : IN  std_logic_vector(size_k-1 downto 0);  --F,C son los indices de
                                                     --la matriz que se forma
       C : IN  std_logic_vector(size_k-1 downto 0);  --con los valores en
                                                     --torno al pixel(A,B)
       K_radio : IN  std_logic_vector(sz_rad-1 downto 0);
       resA : OUT std_logic_vector(sz_indx-1 downto 0);
       resB : OUT std_logic_vector(sz_indx-1 downto 0);     --B' indices de
                                                            --la submatriz
       sal_ready : OUT  STD_LOGIC);
```

```
end sub_mat;
--------------------------------------------------------------------------
architecture Behavioral of sub_mat is

begin
u2:process(clk,reset,enable,F,C,A,B)
variable A_reg:std_logic_vector(A'range);
variable B_reg:std_logic_vector(B'range);
variable N_aux:std_logic_vector(N'range);
variable M_aux:std_logic_vector(M'range);
variable L:std_logic_vector(size_k-1 downto 0);
variable aux : bit;

begin
if (reset='1')then
       A_reg:=(others => '0');
       B_reg:=(others => '0');
       resA <=(others => '1');
       resB <=(others => '1');
       aux := '1';
       sal_ready<='0';
elsif (rising_edge(clk)) then
       if     enable='1' then
              case aux is
              when '1' =>
                     L:='0' & K_radio;   --el mismo tamaño que F y C
                     A_reg := A-L+F;     --Filas, kernel con indices desde 0,0
                     B_reg := B-L+C;     --Columnas,matriz de la imagen desde
                                  --0,0 la imagen podria iniciar en (1,1)
                     N_aux := N-1;       --para una imagen de 128x128 maxima
                     M_aux := M-1;
                     if (A_reg(A_reg'high) = '1') then
                            A_reg := (others=>'0');                --A'
                     elsif (A_reg >= N_aux(A_reg'range)) then
                            A_reg := N_aux(A_reg'range);
                     end if;
                     if (B_reg(B_reg'high) = '1') then
                            B_reg := (others=>'0');                --B'
                     elsif (B_reg >= M_aux(B_reg'range)) then
                            B_reg := M_aux(B_reg'range);
                     end if;
                     resA <= A_reg;
                     resB <= B_reg;
                     sal_ready<= '0';
                     aux := '0';
              when '0' =>
                     sal_ready <= '1';
              end case;
       else aux := '1';
                sal_ready <= '0';
       end if;
end if;
end process;
end Behavioral;
```

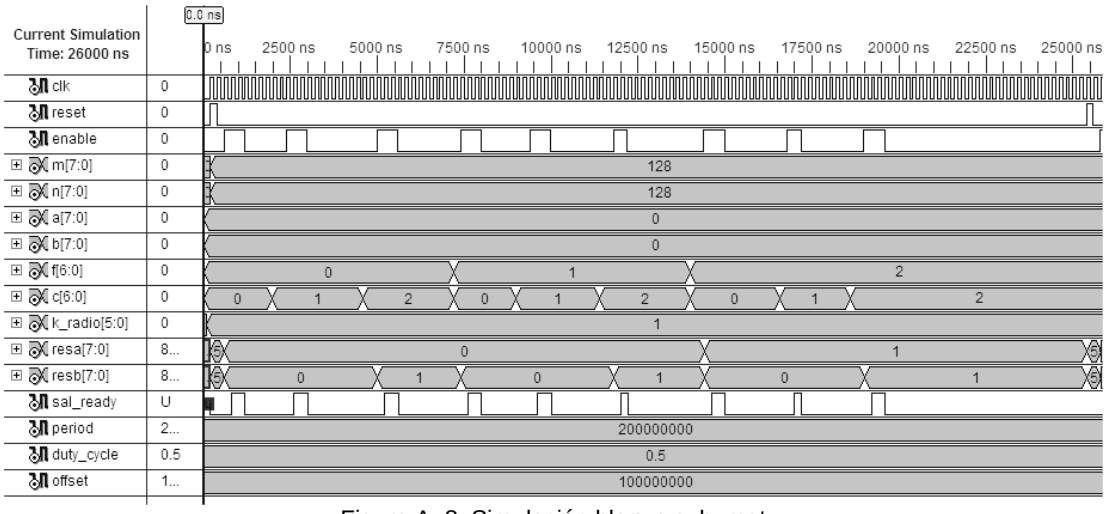

Figura A. 8. Simulación bloque sub_mat

Listado bloque kernel_var.

```
----------------------------------------------------------------------------
------
--genera los valores del kernel, funciona con (F,C) desde (0,0)
--es un circulo de diametro 2*k_radio+1
--la salida permance igual hasta el siguiente pulso de enable
--si enable se deja en 1, la salida cambia con el cambio en las entradas
----------------------------------------------------------------------------
------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity kernel_var is
generic (size_k : integer:=7;    --para que F,C lleguen hasta 64x54 o 65x65
       sz_rad: integer:=6);      --radio es la mitad del kernel
Port ( clk : IN std_logic;
       reset:IN std_logic;
       enable:IN std_logic;
       F : IN  std_logic_vector(size_k-1 downto 0);--F,C submatriz
       C : IN  std_logic_vector(size_k-1 downto 0);
       k_radio : IN  std_logic_vector(sz_rad-1 downto 0);
       K_dato: OUT std_logic
       );
end kernel_var;
----------------------------------------------------------------------------
architecture Behavioral of kernel_var is

begin
process (clk,reset,enable,F,C)
variable rdio : std_logic_vector(size_k-1 downto 0);
variable rdio2 : std_logic_vector((2*size_k)-1 downto 0);
variable radio2 : std_logic_vector((2*size_k)-1 downto 0);
variable centro : std_logic_vector(size_k-1 downto 0);
variable K_reg : std_logic_vector(size_k-1 downto 0);
variable C_aux : std_logic_vector(size_k-1 downto 0);
```

```
variable F_aux : std_logic_vector(size_k-1 downto 0);
variable C1 : std_logic_vector(size_k-1 downto 0);
variable F1 : std_logic_vector(size_k-1 downto 0);
variable aux : bit;

begin
if reset = '1' then
       K_dato <= '0';
       aux := '1';

elsif rising_edge(clk) then
       if enable = '1' then
              case aux is
              when '1' =>
                     C1 := C + 1;        --punto inicial se define como (1,1)
                     F1 := F + 1;        --F,C comienza en (0,0)
                     rdio := '0' & k_radio;    --el mismo tamaño
                     rdio2 := rdio*rdio;       --que F y C
                     centro := '0' & k_radio + 1;
                     if centro > C1 then
                            C_aux := centro - C1;
                     else
                            C_aux := C1 - centro;
                     end if;
                     if centro > F1 then
                            F_aux := centro - F1;
                     else
                            F_aux := F1 - centro;
                     end if;
                     radio2 := C_aux*C_aux + F_aux*F_aux;
                     if radio2 <= rdio2 then
                            k_dato <= '1';
                     else
                            k_dato <= '0';
                     end if;
                     aux:='0';
              when '0' =>
                     null;
              end case;
       else aux:='1';
       end if;
end if;
end process;
end Behavioral;
```

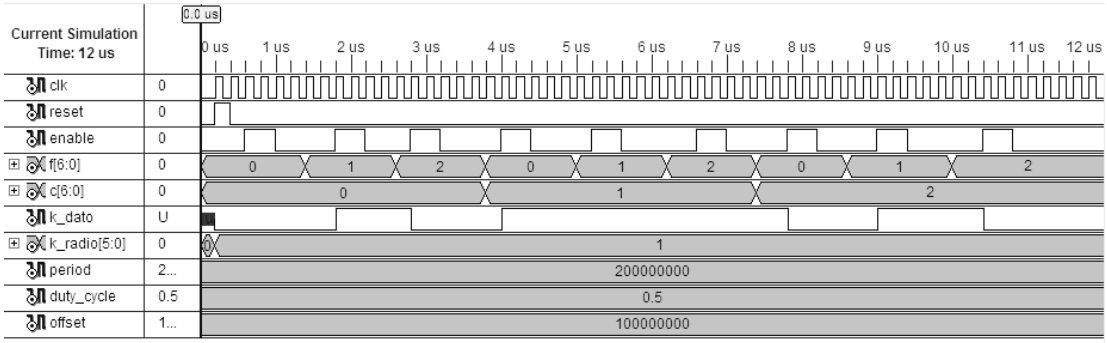

Figura A. 9. Simulación bloque kernel_var.

Listado bloque oper1.

```
--------------------------------------------------------------------------------
------
--efectua la operacion pixel a pixel
--sel_op selecciona que operacion se va a realizar
--si enable se hace 0 se borran el registro de la suma de productos
--la de area permanece sin cambio hasta que se hace clr_area = 1
--Oper_Ready dura lo mismo que suma
--------------------------------------------------------------------------------
------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

entity oper1 is
       generic (sz_area : integer:=14);--para un area total de 16384
       port ( clk : in  Std_Logic;
              reset : in  Std_Logic;
              enable : in  Std_Logic;
              clr_area : in  Std_Logic;
              suma : in  Std_Logic;
              last_pxl : in  Std_Logic;
              sum_SbMat : in  Std_Logic;
              sel_op : in  Std_Logic;
              op_1   : in  Std_Logic;                --viene de la memoria.
              op_2 : in  Std_Logic;                  -- viene del kernel
              area : out  Std_Logic_vector(sz_area-1 downto 0);
              Res_out : out Std_Logic;
              Oper_Ready : out  Std_Logic); -- resultado de 16bits
end entity;
--------------------------------------------------------------------------
architecture Behavioral of oper1 is

begin
process(clk,reset,enable,suma,op_1,op_2,sel_op,clr_area)
variable sum_mult : std_logic;
variable sum_mult1 : std_logic;
variable oper_1 : std_logic;
variable oper_2 : std_logic;
variable cnt_area : integer range 0 to 16383; --imagen 128x128
variable aux : bit;

begin
if reset = '1' then
       Res_out <= '0';
       If sel_op = '1' then
              sum_mult1 := '0';
              sum_mult := '0';
       else sum_mult := '1';
              sum_mult1 := '1';
       end if;
       oper_1 := '0';
       oper_2 := '0';
       aux := '1';
       cnt_area := 0;
```

```
	area <= (others => '0');
	Oper_Ready <= '0';
elsif rising_edge(clk) then
	if clr_area = '1' then
		cnt_area := 0;
	end if;
	if enable = '1' then
		oper_1 := op_1;
		oper_2 := op_2;
		if suma = '1' then
			case aux is
			when '1' =>
				if oper_2='1' then
					if sel_op = '1' then
						sum_mult1 := sum_mult or oper_1;
										--dilatacion
					else sum_mult1 := sum_mult and oper_1;
										--erosion
						--pecstrum erosion - dilatacion
					end if;
				end if;
-----------------------------------------------------------------------
			--rutina para area de imagen
				if sum_SbMat = '1' then
					if sum_mult1 = '1' then
						cnt_area := cnt_area + 1;
					end if;
					if last_pxl = '1' then
			area <= conv_std_logic_vector(cnt_area, area'length);
					end if;
				end if;
-----------------------------------------------------------------------
				Res_out <= sum_mult1;
				aux := '0';
				sum_mult := sum_mult1;
			when '0' =>
				Oper_Ready <= '1';
			end case;
		else aux := '1';
			Oper_Ready <= '0';
		end if;
	else Res_out <= '0';
		aux := '1';
		Oper_Ready <= '0';
		if	sel_op = '1' then
			sum_mult1 := '0';
			sum_mult := '0';
		else sum_mult := '1';
			sum_mult1 := '1';
		end if;
	end if;
end if;
end process;
end Behavioral;
```

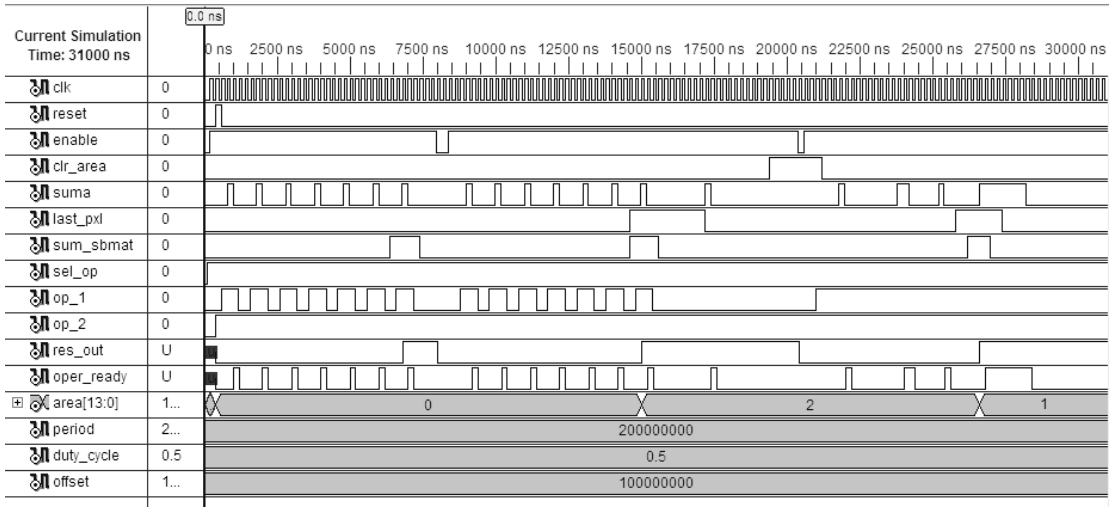

Figura A. 10. Simulación bloque oper1.

Listado bloque LeeGrab_AddrRam2

```
----------------------------------------------------------------------------
------
--genera la dirección de la ram donde se lee o se guarda el pixel
--en base a los subindices de la matriz (A,B)
--conserva la salida con enable = 0
--addr_ready dura lo mismo que enable
--enable debe estar en alto dos ciclos
----------------------------------------------------------------------------
------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity LeeGrab_AddrRam2 is
	generic (sz_indx: integer:=8;
		dim_ram : integer :=14);
	Port ( clk : in STD_LOGIC;
		Reset : in STD_LOGIC;
		Enable : in STD_LOGIC;
		lee_grab: in STD_LOGIC;
		sel_ram : in STD_LOGIC;
		N : in  STD_LOGIC_VECTOR (7 downto 0);  --numero de columnas
		A_lee : in  STD_LOGIC_VECTOR (sz_indx-1 downto 0);
		B_lee : in  STD_LOGIC_VECTOR (sz_indx-1 downto 0);
		A_grab : in  STD_LOGIC_VECTOR (sz_indx-1 downto 0);
		B_grab : in  STD_LOGIC_VECTOR (sz_indx-1 downto 0);
		addr_ready : out STD_LOGIC;
		addr : out  STD_LOGIC_VECTOR (dim_ram downto 0));
end LeeGrab_AddrRam2;
----------------------------------------------------------------------------
architecture Behavioral of LeeGrab_AddrRam2 is

begin
addr_p:process(clk,reset,lee_grab,enable,sel_ram)
variable addr_aux : STD_LOGIC_VECTOR (N'length+sz_indx-1 downto 0);
variable AB : std_logic;
variable aux : bit;
```

```
begin
if (reset='1') then
addr<=(others=>'0');
addr_ready <= '0';
aux := '1';
elsif (rising_edge(clk)) then
	if (enable='1') then
		case aux is
		when '1' =>
			if lee_grab = '1' then
				addr_aux := (N*A_lee) + B_lee;
							--para subindices A,B=0,0
			else addr_aux := (N*A_grab) + B_grab;
							--addr_aux := N*(A-1)+B-1;
			end if;			--para subindices A,B=1,1
			addr <= sel_ram & addr_aux(dim_ram-1 downto 0);
			aux := '0';
			addr_ready <= '0';
		when '0' =>
			addr_ready <= '1';
		end case;
	else
		addr_ready <= '0';
		aux := '1';
	end if;
end if;
end process;
end Behavioral;
```

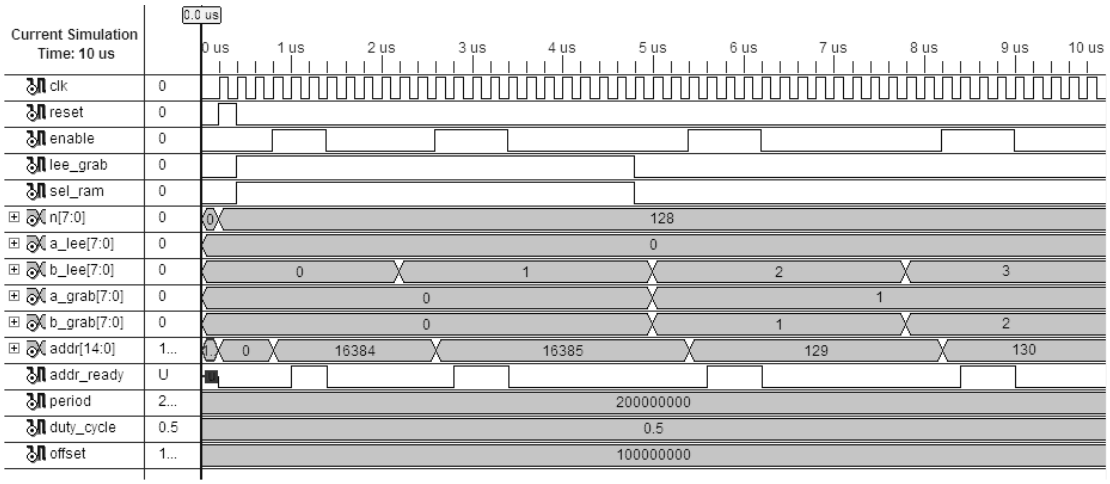

Figura A. 11. Simulación bloque LeeGrab_AddrRam2

Listado bloque ctrl_proc1.

```
----------------------------------------------------------------------------
------
--controla el pocesado de la imagen
--manda las señales de enable
----------------------------------------------------------------------------
------
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ctrl_proc1 is
generic (sz_rad : integer := 6); --6 para un radio de 32 con kernel de 65x65
Port ( clk: in STD_LOGIC;
       Reset : in STD_LOGIC;
       Enable : in STD_LOGIC;
       ini_proc : in STD_LOGIC;  --para iniciar el procesado de la imagen
       pxl_listo : in STD_LOGIC;        --cont_pxl
       last_pxl : in STD_LOGIC;         --cont_pxl
       SbMat_listo: in STD_LOGIC;       --ctrl_submat
       SbMat_Fin : in STD_LOGIC;        --ctrl_submat
       op_ready  : in STD_LOGIC;
       vnt_ready : in STD_LOGIC;
       pxl_grab : in STD_LOGIC;         --indica que el pixel ya se grabo
       LG_RW : out std_logic;           --lee_grab
       sel_mem : out std_logic;         --seleccion memoria
       sel_op : out std_logic;   -      --selecciona operación
                                        --(dilatacion, erosion)
       En_AreaRam: out std_logic;
       En_oper : out std_logic;
       Clr_Area : out std_logic;        --borra el area en oper1
       En_SubMat : out std_logic;       --sub_mat
       En_pxls : out std_logic;         --cont_pxl
       next_pxl : out std_logic;        --cont_pxl
       En_ctrlSbMat: out std_logic;     --ctrl_submat
       sig_K : out std_logic;           --ctrl_submat
       En_DirRam : out std_logic;       --para grabar el resultado en la ram
       k_radio : out std_logic_vector(sz_rad-1 downto 0);
       fin_proc : out std_logic
       );
end ctrl_proc1;
----------------------------------------------------------------------------
architecture Behavioral of ctrl_proc1 is
type ctrl is (Inicio, Inicio2, First_pxl, EnSbMat, leer, Sig_sub, Sig_pxl,
mod_grab, Graba, Fin, Fin2);
signal CtrlProc : ctrl;

begin
process(clk,reset)
variable En1: std_logic;
variable s_mem : std_logic;
variable k_rnel : integer range 0 to 32;

begin
if (reset='1') then
       En_oper <= '0';
       En_SubMat <= '0';
       En_pxls <= '0';
       En_ctrlSbMat <= '0';
       En1 := '0';
       En_AreaRam <= '0';
       k_rnel := 1;
       LG_RW <= '0';
       Clr_Area <= '1';
       next_pxl <= '0';
       sig_K <= '0';
       fin_proc <= '0';
       En_DirRam <= '0';
```

```
        sel_op <= '0';                  --inicia con erosion
        k_radio <= conv_std_logic_vector(k_rnel,k_radio'length);
        s_mem := '0';             --selecciona bloque de memoria - inicio
                                  --parte baja lee(0); parte alta graba(1)
        sel_mem <= '0';
        CtrlProc <= Inicio;
elsif (rising_edge(clk)) then
        if (enable = '1') then
              Clr_Area <= '0';          --solo dura un pulso de reloj
              En_AreaRam <= '0';        --solo dura un pulso de reloj
              case CtrlProc is
              when Inicio =>
                    if ini_proc = '1' then
                          CtrlProc <= Inicio2;
                    else CtrlProc <= Inicio;
                    end if;
              when Inicio2 =>
                    Clr_Area <= '1';                --borra el area
                    En_pxls <= '1';                 --habilita cont_pxl
                    En_ctrlSbMat <= '1';
                    En_oper <= '1';
                    CtrlProc <= First_pxl;
              when First_pxl =>
                    LG_RW <= '1';       --lee_grab modo leer; ram modo leer
                    sel_mem <= s_mem;   --selecciona la parte de la ram lee
                    next_pxl <= '1';
                    CtrlProc <= EnSbMat;
              when EnSbMat =>
                    sig_K <= '1';
                    if pxl_listo = '1' then
                          if SbMat_listo = '1' then
                                En_SubMat <= '1';
                                CtrlProc <= leer;
                          else CtrlProc <= EnSbMat;
                          end if;
                    else CtrlProc <= EnSbMat;
                    end if;
              when leer =>
                    if vnt_ready = '1' then
                          En_DirRam <= '1';   --enable de LeeGrab_AddrRam
                          CtrlProc <= Sig_sub;
                          sig_K <= '0';
                    else CtrlProc <= leer;
                    end if;
              when Sig_sub =>
                    if op_ready = '1' then
                          if SbMat_Fin = '1' then
                                En_ctrlSbMat <= '0';--borra los contadores
                                CtrlProc <= mod_grab;
                          else CtrlProc <= EnSbMat;
                          end if;
                          En_SubMat <= '0';
                          En_DirRam <= '0';
                    else CtrlProc <= Sig_sub;
                    end if;
              when mod_grab =>
                    LG_RW <= '0';--lee_grab modo grabar; ram modo grabar
                    sel_mem <= not(s_mem);--seleciona memoria donde graba
                    CtrlProc <= Graba;
              when Graba =>
                    En_DirRam <= '1';         -- enable de LeeGrab_AddrRam
```

```
                    if pxl_grab = '1' then
                          if last_pxl = '1' then
                                En_pxls <= '0';
                                CtrlProc <= Fin;
                          else En_ctrlSbMat <= '1';
                                CtrlProc <= Sig_pxl;
                          end if;
                          En_DirRam <= '0';
                          En_oper <= '0';     --borra registros de oper
                                              --ultimo resultado se conserva
                          next_pxl <= '0';
                    else CtrlProc <= Graba;
                    end if;
             when Sig_pxl =>
                    sel_mem <= s_mem;   --selecciona la memoria donde se lee
                    LG_RW <= '1';       --lee_grab modo leer; ram modo leer
                    next_pxl <= '1';
                    En_oper <= '1';
                    CtrlProc <= EnSbMat;
             when Fin =>
                    if s_mem = '1' then
                          k_rnel := k_rnel + 1;
                          sel_op <= '0';            --selecciona erosion
                          En_AreaRam <= '1';  --guarda el area
                    else sel_op <= '1';       --selecciona dilatacion
                    end if;
                    if k_rnel <= 24 then     --32 aperturas, kernel de 65x65
                          s_mem := not(s_mem);
                    k_radio <= conv_std_logic_vector(k_rnel,k_radio'length);
                          CtrlProc <= Inicio2;
                    else fin_proc <= '1';
                          CtrlProc<=Fin2;
                    end if;             --conserva el ultimo que proceso
             when Fin2 =>
                    En_pxls <= '0';
                    En_oper <= '0';
                    En_SubMat <= '0';
                    En_ctrlSbMat <= '0';
                    CtrlProc<=Fin2;
             when others => null;
       end case;
       end if;
end if;
end process;
end Behavioral;
```

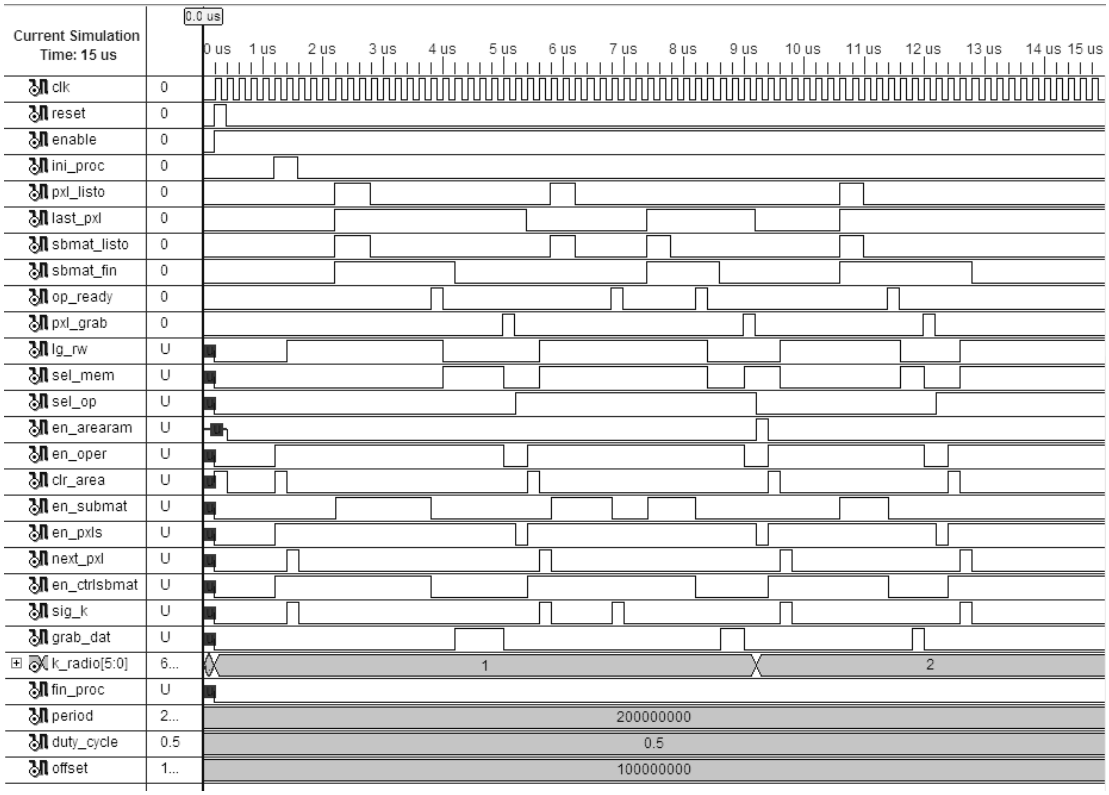

Figura A. 12. Simulación bloque ctrl_proc1

Índice de tablas y figuras.

Tablas