Centro de Investigación y de Estudios Avanzados del
Instituto Politécnico Nacional
Departamento de Ingeniería Eléctrica
Sección de Computación

Diseño en FPGA de un circuito comparador de imágenes

Tesis que presenta:

Miguel Angel Sánchez Martínez

Para obtener el grado de:

Maestro en Ciencias

En la Especialidad de:

Ingeniería Eléctrica

Director de Tesis:

Dr. Adriano de Luca Pennacchia

México,D.F. Julio 2005

Resumen

La inspección visual de tarjetas de circuito impreso, realizada en las primeras etapas de su ensamble, permite mejorar la calidad de los objetos producidos y aumentar la producción. El depósito de soldadura en pasta, utilizado en tarjetas con circuitos de soldadura superficial, es una de tales etapas. Sin embargo, su inspección visual realizada por personas está sujeta a errores, por ello se utilizan sistemas automáticos para realizar tal tarea. En su mayoría, estos sistemas están basados en un algoritmo de procesamiento de imagen ejecutándose en una procesador convencional. Esto hace díficil cumplir con los requerimientos de tiempo que exige la industria moderna.

Esta tesis presenta un sistema para analizar imágenes de circuitos impresos con pastas depósitadas, basado en un algoritmo de procesamiento de imágenes por textura implementado en hardware, cuyo propósito es ofrecer un tiempo de cálculo aceptable. El algoritmo extrae una serie de valores de la región analizada y los compara con los obtenidos de una imagen de referencia, la cual corresponde a la de una tarjeta sin defectos. Cualquier diferencia entre las series de valores indica una variación entre las imágenes analizadas, y por tanto una falla en el depósito de las soldaduras.

Las imágenes capturadas son divididas en cuadros disjuntos de 40x40 píxeles. A cada uno de estos segmentos se les aplica una serie de operaciones que nos permite obtener un par histogramas, calculados a partir de las variaciones de brillo y contraste, los cuales son representativos de la textura. Estos histogramas son comparados con los respectivos de una imagen de referencia, de acuerdo a la posición en la imagen de la región analizada. Una diferencia entre los histogramas es producida por fallas en el depósito de soldadura, que provocan cambios en la textura de la imágen analizada. Con el sistema propuesto, el tiempo de cálculo necesario para comparar una imagen en escala de grises de 640x480 píxeles es de 24 ms, suficiente para aplicarlo en el análisis industrial de tarjetas.

Abstract

Printed circuit board inspection, carried out in first stages of assembly process, allows us to improve quality of produced objects as well as to increase their production. Solder paste deposit, which is used in surface mount technology, is one of such stages. However, this inspection performed by humans is subject to mistakes. That is why automatic inspection systems have been used to perform such labor. Most of these systems are based in an image processing algorithm being executed with a conventional processor. This makes dificult to fulfill modern indutry time requirements.

This thesis presents a system to analyze images of solder paste deposited on printed circuit boarrds. This system uses a texture based image processing algorithm, implemented in hardware, with the purpose of achieving an accepted processing time. This algorithm extracts a series of values from the analyzed region and then compares them with those obtained from a reference image, which corresponds to a non-defective printed circuit board. Any difference between these series of values indicates that these images are different, and therefore there is bad solder paste deposit.

Captured images are divided in 40 x 40 pixels non-overlapping regions. Some operations are applied to each one of these segments in order to obtain a pair of histograms, computed from bright and contrast variations, which are texture representative. These histogramas are compared with those obtained from a reference image, according to the position of the analyzed region. Any difference between these histograms is produced by defects in solder paste deposit that causes changes in texture. With the proposed system, the necessary time to compare a 640 x 480 pixels grayscale image is 24 ms, enough for printed circuit board industrial analysis.

Agradecimientos

A familiares y amigos por su apoyo.

Al Consejo Nacional de Ciencia y Tecnología por el apoyo otorgado para realizar esta tesis.

Al Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, por darme la oportunidad de realizar mis estudios de maestría.

A los profesores de la sección de computación por las enseñanzas recibidas.

A los profesores que realizaron la revisión del documento final, Dr. Luis Gerardo de la Fraga y Dr. Francisco Rodríguez Henríquez de la sección de computación, y al Dr. Vicente Parra Vega de la sección de mecatrónica.

Un agradecimiento especial al Dr. Ernst Kussul y a la Dra. Tatiana Baidyk del Centro de Ciencias Aplicadas y Desarrollo Tecnológico de la Universidad Nacional Autónoma de México, por los comentarios hechos para realizar esta tesis.

Igualmente agradezco al personal administrativo de la sección de computación y de la biblioteca por su atención y ayuda.

Índice general

Capítulo 1

Introducción

El proceso de ensamble de tarjetas de circuito impreso, con dispositivos de soldadura superficial, consta de varias etapas. Primero se encuentra la elaboración de las tarjetas, la cual generalmente es hecha por alguna empresa diferente a la que realiza el ensamble. Después debe de ser depositada soldadura en pasta sobre el circuito impreso. Esta pasta es colocada, con la ayuda de una mascarilla, sobre las terminales donde serán colocados los dispositivos. Posteriormente son montados los dispositivos sobre las tarjetas, y estas son colocadas dentro de un horno, para que se funda la soldadura y los dispositivos queden fijos. Finalmente, las tarjetas estan listas para su uso.

Para las empresas es importante la calidad de los productos que se fabrican, y esta queda reflejada en la satisfacción de los clientes finales que adquieren los productos. Cualquier falla de estos es perjudicial para la empresa. Por ello es importante la elaboración de estrategias para asegurar la calidad, con el objetivo de evitar que productos defectuosos lleguen al usuario final. En la industria manufacturera se pueden reconocer varias etapas sucesivas en la fabricación de algún producto, como las descritas al principio para el caso de las tarjetas. Una estrategia para asegurar la calidad del producto final es asegurar la calidad de los productos intermedios, o sea, los obtenidos en cada una de las etapas de producción. Esto nos permite ver la importancia que tienen las primeras etapas en la fabricación de un producto, y desde un principio evitar la salida de alguno defectuoso.

Para el caso de la producción de tarjetas de circuito impreso con dispositivos superficiales, una de las primeras etapas es la colocación de soldadura en pasta. Se sabe que alrededor del 60 % de fallas en tarjetas, con dispositivos de soldadura superficial, se deben a problemas con el depósito de soldadura en pasta [11]. Un buen depósito sobre el circuito impreso asegura una buena formación de las soldaduras, evitando fallas como puentes de soldadura, soldadura dispersa o falta de soldadura en la unión. En caso de que se presente alguna falla en el depósito de pasta, solo se tiene que retirar la tarjeta afectada para limpiarla y utilizar la nuevamente, pero sobre todo se evita que se propage la falla.

Para detectar este tipo de fallas se hace uso de la inspección visual de las tarjetas, la que puede ser realizada de forma manual o automática. La inspección manual es realizada por una persona que se encarga de revisar la tarjeta en busca de fallas. El problema con este tipo de inspección es que con el tiempo la persona se cansa, su rendimiento disminuye y la revisión depende de factores subjetivos. En la inspección automática se eliminan estos problemas por medio de la utilización de un equipo electrónico que no se cansa y siempre utiliza el mismo criterio [7].

En su mayoría, los equipos usados para la inspección visual automática está basado en un procesador convencional ejecutando algún algoritmo de procesamiento de imágenes [6]. Sin embargo, el precio de estos equipos es muy elevado para las empresas mexicanas dedicadas al ramo. Además de esto, y debido a la gran cantidad de cálculos que implica manejar imágenes, los tiempos de ejecución del algoritmo hacen díficil cumplir con los requerimientos de tiempo. Es necesario mencionar que para las empresas, además de buscar la calidad de los productos, también es importante conseguir el nivel de producción deseado. Se ha propuesto que para cumplir con estos requerimientos, un sistema de inspección debe analizar las imágenes en tiempo real [7].

En esta tesis se propone una arquitectura de un circuito comparador de imágenes, el cual será usado en un sistema automático de inspección visual, para tarjetas de circuito impreso con dispositivos de soldadura superficial. Con esta arquitectura se pretende re-

ducir los tiempos de procesamiento y los costos de los sistemas de inspección actuales. Los tiempos de ejecución se pueden reducir ya que el algoritmo de procesamiento de imagen es implementado en hardware, para ello el sistema hace uso de un dispositivo lógico programable, específicamente un FPGA (Field Programmable Gate Array). El uso del FPGA nos permite obtener un sistema compacto y más barato que los sistemas de inspección actuales, ya que toda la parte electrónica de control y procesamiento de imagen puede quedar alojado en un solo circuito. La programación del FPGA fue realizada utilizando el lenguaje de programación VHDL, lo que nos permitió obtener un diseño rápido del sistema.

El algoritmo de procesamiento de imagen implementado se basa en la comparación de dos imágenes, una correspondiente al de una tarjeta sin fallas (imagen base), y la otra a la imagen de la tarjeta a analizar (imagen de prueba). La comparación se realiza tomando sucesivamente segmentos de 40x40 píxeles, para obtener un histograma de brillo y contraste que representa las características de textura del segmento analizado. Este histograma es característico de cada segmento de la imagen, de tal forma que hay un histograma para el circuito impreso, otro para la pasta, otro para la terminal, y otros para sus combinaciones. El histograma del segmento de la imagen de prueba es comparado con su correspondiente de la imagen base. Una variación en el histograma indica diferencias en las imágenes analizadas, y por tanto fallas en el depósito de la soldadura en pasta. Por ejemplo, en la imagen base se indica que la región analizada debe tener textura de circuito impreso. Sin embargo, por una falla en el depósito, en la región analizada de la imagen de prueba se encontró textura de pasta, la que produce un histograma diferente.

Para el presente trabajo, las imagenes capturadas están en 256 tonos de grises. La imagen completa es segmentada para formar cuadros disjuntos de 40x40 píxeles. Estos segmentos son la entrada del circuito analizador desarrollado para la presente tesis. Este circuito tiene la tarea de generar los histogramas de brillo y contraste del segmento recibido, con el fin de compararlo con el histograma del segmento correspondiente de la imagen base. Para

incrementar la velocidad de procesamiento, se propone el uso de varios de estos circuitos analizadores para diferentes segmentos de la imágen, de tal forma que pudieran trabajar en paralelo. Para propósito de prueba, en caso de que el sistema encuentre una falla en el depósito de soldadura, ésta será indicada por medio de una señal de salida del circuito diseñado.

El propósito de esta tesis es determinar el tiempo de cálculo para procesar un segmento de imagen, usando la unidad analizadora básica desarrollada, así como el espacio de FPGA necesario. Con esta información es posible estimar el tiempo necesario para analizar una imagen capturada, y el espacio de FPGA que ocuparía el sistema de análisis de imagen propuesto.

El resto de la tesis esta organizada como se muestra a continuación:

Capitulo 2. Inspección de tarjetas de circuito impreso. En esta parte se explica el proceso industrial de soldadura, aplicado a dispositivos SMT (Surface Mount Technology). Se explican los diferentes métodos usados para realizar la inspección, y se justifica el método usado para el sistema desarrollado.

Capitulo 3. Algoritmo para comparación de imágenes. De acuerdo con el tipo de imágenes que se desean analizar, el método escogido hace uso de la textura, la cual es representada por las características de brillo y contraste. En este capítulo se hace una explicación de varios métodos para análisis por textura, así como las ventajas del método usado.

Capítulo 4. Síntesis de circuitos FPGA por medio de VHDL. Aquí se dada una breve explicación del lenguaje de programación VHDL y su uso para síntesis de FPGA's. También hay una explicación del flujo de diseño que hay que seguir para obtener un circuito funcionando, desde las especificaciones del sistema hasta la programación del FPGA.

Capitulo 5. Circuito analizador de segmentos. Aquí se presentan las etapas que componen la parte de procesamiento de imagen del sistema de inspección. La atención se centra en el circuito analizador. Este circuito forma la unidad basica de procesamiento. El sis-

tema completo consta de un grupo de estos circuitos, cada uno procesando un segmento de la imagen analizada y todos actuando de forma paralela. En este capítulo se explica a detalle el diseño del circuito analizador.

Capítulo 6. Resultados y Conclusiones. Aquí aparecen los resultados obtenidos usando el circuito con las imagenes de prueba. Se presenta un resumen de los objetivos logrados con el presente trabajo, así como las tareas pendientes para obtener el sistema completo funcionando.

Capítulo 2

Inspección de tarjetas de circuito impreso

En este capítulo se da una explicación del proceso industrial de soldadura de tarjetas de circuito impreso. La atención se centra en el proceso de soldadura de tarjetas con elementos de soldadura superficial o SMT. Este tipo de tecnología es la que predomina actualmente, ya que este tipo de tarjetas las podemos encontrar en diferentes equipos electrónicos, como computadoras, radios, televisiones, camáras digitales, entre otros. Por ello, el sistema de inspección que se está desarrollando analizará este tipo de tarjetas.

2.1. El proceso de soldadura

La soldadura en pasta es utilizada en la industria para la colocación de los dispositivos SMT. Consiste de una mezcla de pequeñas esferas de soldadura con flux o pasta para soldar. El proceso inicia con el depósito de la soldadura en pasta sobre la tarjeta, únicamente en las áreas donde harán contacto las terminales de los dispositivos. Como se muestra en la figura 2.1, este depósito es realizado con la ayuda de una mascarilla, la cual tiene orificios que coinciden con las terminales de la tarjeta.

Después de que ha sido colocada la mascarilla sobre la tarjeta, se hace pasar una

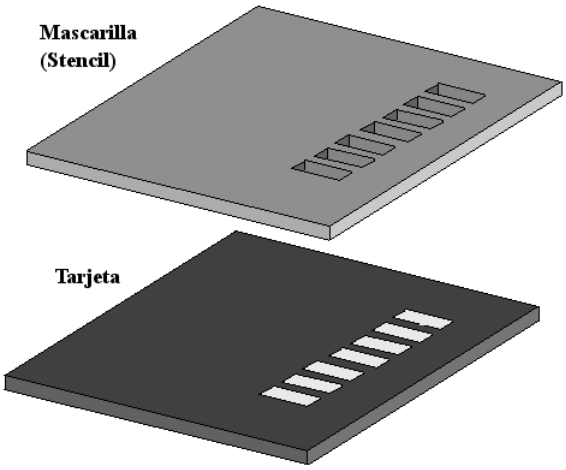


Figura 2.1: Mascarilla para depósito de soldadura en pasta.

navaja de goma la cual va distribuyendo la pasta sobre los orificios de la mascarilla, como se muestra en la figura 2.2. Posteriormente, los diferentes dispositivos son colocados en su respectiva localidad. Después las tarjetas son introducidas en un horno, con lo que se logra que la soldadura se derrita y, con la ayuda de la pasta, formen las uniones de las terminales de los dispositivos con la tarjeta [12]. Cabe mencionar que el proceso descrito es realizado de manera automática, de aqui la importancia de la inspección para verificar el correcto depósito de soldadura.

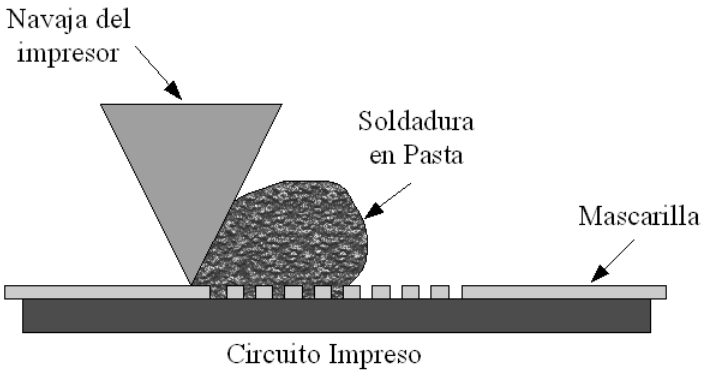


Figura 2.2: Distribución de la soldadura en pasta.

Durante este proceso se pueden presentar varios problemas, que son la causa de un mal depósito de la soldadura en pasta. Por ejemplo, la mascarilla puede estar rota y con ello

provocar que la pasta se deposite en otras áreas, dando origen a posibles cortocircuitos. Otro problema se presenta al momento de retirar la mascarilla, ya que una parte de la soldadura depósitada es retirada al mismo tiempo, lo que trae como consecuencia que sea poca la cantidad en la terminal. Puede ocurrir tambien que haya una mala distribución de la soldadura en pasta en el rodillo, de tal forma que al momento de pasarlo sobre la mascarilla algunos orificios no reciben la cantidad de pasta necesaria. Otra causa puede ser el utilizar por error una mascarilla incorrecta, por lo que el depósito de la soldadura queda sobre otras áreas de la tarjeta.

Estos posibles problemas dan como resultado los errores de depósito mostrados en la figura 2.3.

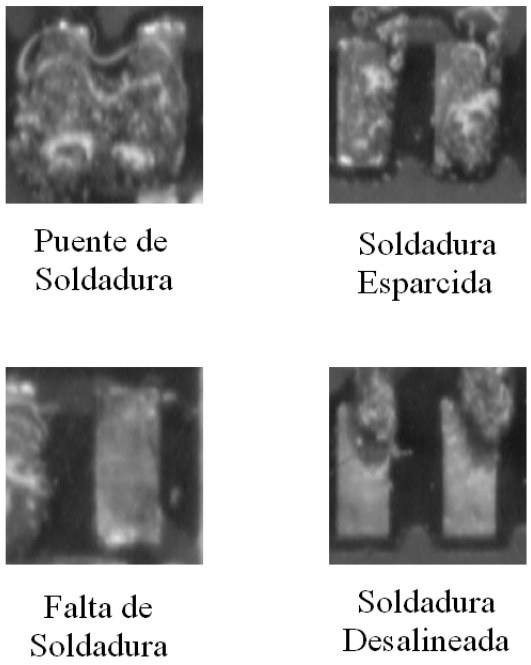


Figura 2.3: Tipos de fallas en el depósito de soldadura.

Para propósitos del presente trabajo, y puesto que el análisis de imagen es en dos dimensiones, solo se consideran fallas en la colocación de la pasta. Para determinar la cantidad de soldadura depósitada es necesario realizar un análisis en 3 dimensiones, trabajo que será efectuado en proyectos futuros.

Con las anteriores imágenes como base, se tienen los elementos para poder definir

el método a utilizar para realizar la inspección. Pero antes, es necesario dar una breve introducción a los diferentes formas de realizar la inspección.

2.2. Métodos de inspección

Llamaremos inspección de tarjetas al proceso de determinar si una tarjeta presenta alguna falla, la cual tiene la posibilidad de generar un malfuncionamiento de la misma, y por lo mismo del equipo al cual será integrada. Esta inspección es importante, ya que con su ayuda permite mejorar la calidad de los productos finales, ya sea computadoras, televisores, teléfonos, o cualquier otro.

De manera general, los métodos usados para la inspección puden ser de dos tipos [8]:

1. Métodos eléctricos o con contacto.

2. Métodos no eléctricos o sin contacto.

El primer método de inspección es usado sobre todo para tarjetas sin dispositivos. Consiste en aplicar corriente a las pistas para verificar la conexión entre las mismas. También pude ser aplicado con la tarjeta ya terminada, con esto se busca la presencia de cortos entre las terminales de los dispositivos, así como verificar la unión de las terminales con la tarjeta. Sin embargo, si es encontrada una falla con los dipositivos ya soldados, la tarjeta debe ser desechada o reparada, lo cual implica gastos adicionales.

En el segundo método se usa inspección óptica o visual de las tarjetas. Este método nos permite detectar fallas que no es posible detectar con los métodos eléctricos, como exceso de soldadura, fallas estéticas, mala unión, elementos desalineados, etc. Estas fallas pueden no provocar un malfuncionamiento de la tarjeta ya terminada, sin embargo son fuentes potenciales de tal malfuncionamiento. La inspección visual nos permite detectar estas fallas, evitando que las tarjetas defectuosas pasen a las siguientes etapas del proceso de producción, esto con el fin de evitar que afecten la calidad del producto final.

Durante el proceso descrito en la sección 2.1, la inspección puede ser realizada en las siguientes etapas:

1. Depósito de soldadura en pasta. Un buen depósito implica una buena unión de la terminal del dispositivo con la tarjeta, por lo que la inspección en esta etapa evita fallas en etapas posteriores. Además, si es detectada alguna falla, la tarjeta puede ser reutilizada fácilmente.

2. Colocación de dispositivos. En esta etapa se puede verificar si los dispositivos colocados son los correctos y si están bien orientados.

3. Horneado de tarjetas. En esta etapa se pueden revisar las uniones formadas, con el fin de asegurarse que estén firmes.

Para el desarrollo del presente trabajo, se considera que la inspección en las primeras etapas de producción es importante, ya que de esta forma se puede evitar que las fallas se propagen hacia las demás etapas. Siendo el depósito de la soldadura en pasta importante para tener una buena tarjeta, se propone realizar un sistema de inspección visual automático para realizar la inspección. Con la ayuda de este sistema podemos encontrar las siguientes ventajas:

1. Eliminar el factor humano, ya que la inspección realizada por personas está sujeta a fallas de apreciación.

2. Obtener un sistema rápido, ya que el sistema se encuentra implementado en hardware, y no en una computadora convencional como en otros sistemas.

3. Al estar todo el sistema en un FPGA, tiene la característica de ser compacto y económico.

2.3. Características de las imágenes a analizar

Una vez definido el tipo de inspección y la etapa en donde será aplicada la misma, ahora es necesario definir el método para poder realizarla. Para seleccionar el método adecuado es importante conocer las características del producto que se debe inspeccionar. En la figura 2.4 aparece un circuito impreso a ser analizado. Esta imágen es representativa de las imágenes con las que se va a trabajar, y con su ayuda podemos obtener las características necesarias para definir el método a usar.

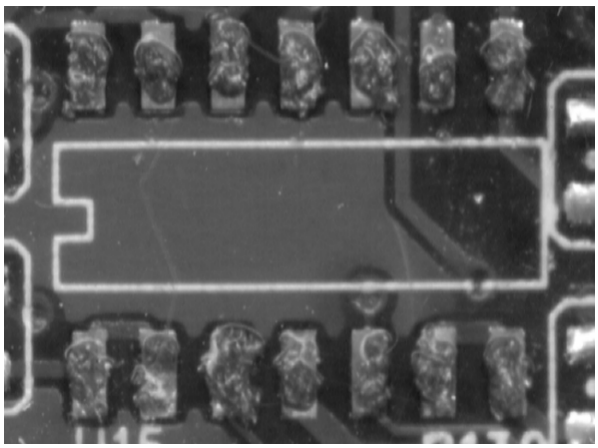


Figura 2.4: Imagen de un circuito impreso con soldadura en pasta.

Tales características son las que se enumeran a continuación:

1. Aparecen 4 áreas representativas. Estas son el circuito impreso, la soldadura en pasta, la terminal y los rótulos.

2. Cada una de estas áreas quedan definidas por sus características de textura.

3. Las imagenes son obtenidas bajo un ambiente controlado. Esto nos asegura que el nivel de iluminación va a ser el mismo para todas las tarjetas, además la variación en la posición será muy pequeño.

4. Para saber si hay una falla en el depósito de soldadura, solo hace falta saber que existe, sin necesidad de especificar el tipo de falla.

De manera especial hay que mencionar el punto referente a la textura, ya que es en base a ésta que podemos hacer la comparación entre dos imágenes. Podemos inspeccionar segmentos correspondientes entre dos imágenes, tratando de encontrar alguna diferencia entre texturas.

En la literatura no hay una definición exacta de textura. Sin embargo, para el caso del presente trabajo, en donde las imagenes están en tonos de grises, el valor de textura para cada pixel de la imágen estará determinado por las características de brillo y contraste. El uso de estas características de la imágen está justificado ya que el punto 3 nos asegura que estos valores serán constantes entre diferentes imágenes.

Finalmente, el punto 4 es el que permite justificar la comparación de imágenes, ya que no necesitamos saber exactamente si hace falta soldadura o si ésta se encuentra en donde no debe estar, por ejemplo. Solo hace falta saber que existe una falla en el depósito de soldadura.

Para el sistema propuesto, se tiene una imagen base, la cual corresponde a la de una tarjeta que se sabe no tiene fallas de depósito. Posteriormente, las imágenes de prueba, que corresponden a las tarjetas a analizar, son comparadas con la imágen base. Esto nos indica que es necesario un algoritmo de comparación de imágenes, que utilice las texturas de las imágenes como criterio de comparación. El resultado de esta comparación debe ser un valor único, que nos indique el nivel de semejanza entre las imágenes. En el capítulo siguiente se explica el algoritmo seleccionado para realizar la inspección.

Capítulo 3

Algoritmo para comparación de imágenes

En el capítulo anterior se definieron las características de las imágenes a analizar. Además, se llegó a la conclusión que un análisis por textura es el más adecuado. En este capítulo se da una breve revisión de los diferentes métodos usados para análisis de textura, así como el funcionamiento del algoritmo de procesamiento de imagen seleccionado.

3.1. Métodos de análisis por textura

Los métodos usados para el análisis por textura pueden ser divididos de la siguiente forma [2]:

- Estructurales
- Estadísticos
- Espectrales

En los métodos estructurales, la textura está representada por medio de primitivas, llamadas microtexturas, las cuales son usadas para formar patrones de textura complejas.

Para obtener una textura es necesario seleccionar las microtexturas, para posteriormente definir las reglas para su colocación.

Para representar la textura, los métodos estadísticos utilizan la distribución de los niveles de grises de una imagen. A partir de ellos es posible obtener valores que representen características de la textura, como son contraste, granularidad, o su suavizado.

Finalmente, los métodos basados en alguna transformada, como la de Fourier, Gabor y Wavelet, representan una imágen en un espacio cuyo sistema de coordenadas tiene una representación que está muy relacionada a las características de textura.

Los métodos estructurales son principalmente utilizados para síntesis de imágenes, y los métodos basados en transformada son computacionalmente complejos. Existen estudios donde se reporta que el uso de métodos estadísticos permite obtener buenos resultados, comparables a los obtenidos usando métodos estructurales o basados en transformada. Por ejemplo Paclick [10] realiza un estudio de varios métodos de análisis de textura, usados para la inspección visual en la industria textil, y usando métodos estadísticos obtuvo buenos resultados, tanto en poder discriminativo como tiempo de procesamiento. Debido a esto, para el desarrollo de la presente tesis se optó por el uso de un método estadístico, con el fin de estudiar su eficiencia con los tipos de imágenes con los que se va a trabajar.

Para la inspección visual de tarjetas de circuito impreso, la tendencia es obtener sistemas que puedan analizar las imágenes en tiempo real [7]. Como fué mencionado en el capítulo 1, el propósito de esta tesis es implementar un algoritmo de procesamiento de imagen en FPGA, con el objeto de poder cumplir con este requerimiento de tiempo. Al momento de implementar un algoritmo en FPGA, hay que tener cuidado de que tal implementación pueda realizarse fácil y adecuadamente. Con esto se quiere decir que el espacio y los recursos del FPGA sean suficientes para poder ejecutar el algoritmo deseado. Los métodos estadísticos generan algoritmos adecuados para hardware, lo que significa que es posible diseñar circuitos que implementan tal algoritmo, de tal forma que puedan ser realizados fácilmente con la ayuda de algún lenguaje de descripción de hardware, como VHDL.

Para el presente proyecto, se necesita una forma de representar la textura, de tal forma que nos permita realizar comparaciones entre imágenes. Con los métodos estadísticos para análisis de textura, podemos obtener una serie de valores que pueden ser usadas para representarla. Este conjunto de valores son los que podemos utilizar para realizar la comparación de imágenes deseada. El uso de histogramas como descriptores de textura ha dado buenos resultados [9] [1], es por ello que se experimentó con histogramas para comprobar su eficiencia con las imágenes de las tarjetas.

3.2. Histogramas de brillo y contraste

Habiendo definido el método de análisis a utilizar, o sea un método estadístico, el siguiente paso fué seleccionar el algoritmo adecuado para realizar la comparación. De acuerdo a las características de las imágenes a utilizar, y que fueron definidas en la sección 2.3, se distinguen cuatro áreas principales, las cuales aparecen a continuación, junto con sus características específicas:

- Área de circuito impreso. Esta es el área correspondiente a la tarjeta del circuito impreso donde no hay ningún dispositivo. Se caracteriza por tener un bajo nivel de brillo, debido a que la tarjeta es cubierta con un barniz obscuro. Además estas áreas son uniformes, lo que nos da un bajo nivel de contraste.

- Área de soldadura en pasta. El gran contraste entre los píxeles vecinos es lo que caracteriza a estas imágenes, generado por las esferas de soldadura y la pasta que lo rodea.

- Área de terminal. Esta es también un área de bajo contraste, debido su uniformidad en la intensidad. Pero a diferencia del área de circuito impreso, presenta mayor brillo, ya que es el área donde la pista queda totalmente libre.

- Área de rótulos. En esta categoría colocamos las impresiones que se realizan en el

circuito impreso. Se caracterizan por tener un brillo mayor que el área de terminal, esto es porque los rótulos son realizados con una pintura de color claro, como el blanco o amarillo.

Después del análisis de las diferentes áreas mencionadas, se puede notar que el brillo y el contraste de las imagenes son las características que necesitamos, ya que con ellas podemos definir el segmento de área que estamos analizando. Además estos valores son los adecuados, ya que se está trabajando con imágenes en tonos de grises.

Debido a lo anterior, podemos establecer que la textura de una imagen va a quedar definida por sus características de brillo y contraste. Para hacer uso de estas características, de tal forma que pudieran ser utilizadas en un algoritmo, se optó por el uso histogramas. Los histogramas han sido usados en diferentes aplicaciones, obteniéndose buenos resultados. Sin embargo, en la mayoría de esos trabajos hay un preprocesamiento de la imágen antes de obtener el histograma, como la aplicación de filtros o el uso de un operador. Para el caso del sistema de inspección que se está desarrollando, tenemos que las imágenes serán obtenidas a partir de un ambiente controlado, con un nivel de iluminación constante y adecuado. Es por ello que en el algoritmo propuesto obtenemos la característica de textura directamente de la imagen capturada. Esto es con el fin de realizar el procesamiento de imagen lo más rápido posible.

Además, en trabajos anteriores se manejan histogramas multidimensionales (aquellos que usan dos o más variables para definir a que intervalo pertenece un elemento)[3] [9]. Para el presente proyecto, y debido a que las tarjetas con pasta depositada tiene solo cuatro tipos de texturas, usamos histogramas unidimensionales (aquellos que usan solo una variable para definir a que intervalo pertenece un elemento), ya que son suficientes para caracterizar la región de la imagen analizada. En el articulo de Baidyk [1] se usan este tipo de histogramas para representar la textura para analizar piezas metálicas. El método propuesto en esta tesis está basado en este artículo.

Para cada imagen vamos a tener dos histogramas, uno para el brillo y otro para el contraste. Para obtener los histogramas, es necesario asignarle a cada pixel dos valores, uno que represente su valor de brillo y otro para el contraste. Para el cálculo de estos valores tomamos en cuenta que cada pixel de la imagen está rodeado por otros ocho píxeles. Llamaremos al pixel cuyos valores de brillo y contraste se desean calcular "pixel central", y a los 8 píxeles que los rodean "píxeles vecinos". Esto se pude ver en la figura 3.1.

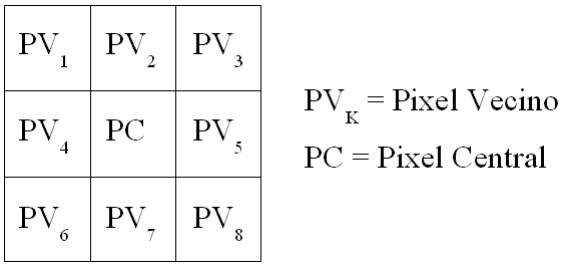


Figura 3.1: Ventana de 3x3 píxeles usado para al cálculo de los valores de brillo y contraste.

En donde:

PC = Valor del pixel central.

PV_k = Valor de los píxeles vecinos, para k = 1 hasta 8.

Para obtener el valor de brillo del pixel central, tomamos el valor promedio de las intensidades de los ocho píxeles vecinos. Esto queda representado por la siguiente fórmula:

$$B_{PC} = \frac{\sum_{k=1}^{8} PV_k}{8} \tag{3.1}$$

Donde:

PV_k = Valor del k-ésimo pixel vecino.

B_{PC} = Valor de brillo para el pixel central.

Aquí hay que notar que solo se usaron las intensidades de los píxeles vecinos, ya que obtenemos la división de un número entero entre una potencia de 2, que puede ser realizada fácilmente en hardware por medio de un registro de corrimiento.

Para el caso del contraste, de los ocho píxeles vecinos, tomamos los que tienen valor mayor o igual al pixel central y se calcula su promedio. Tomando los valores de los píxeles menores al pixel central obtenemos otro promedio. La diferencia entre ambos promedios nos da el valor del contraste. Las siguientes fórmulas nos dan la forma para calcular este valor.

$$promsup = \frac{\sum_{k=1}^{8} PV_k}{numsup} \qquad \forall\ PV_k \geq PC \tag{3.2}$$

$$prominf = \frac{\sum_{k=1}^{8} PV_k}{numinf} \qquad \forall\ PV_k < PC \tag{3.3}$$

Donde:

$numsup$ = Número de píxeles vecinos con valor mayor o igual al pixel central.

$numinf$ = Número de píxeles vecinos con valor menor al pixel central.

PV_k = Valor del k-ésimo pixel vecino.

PC = Valor del pixel central.

$promsup$ = Promedio de valores de los píxeles mayores o iguales al pixel central.

$prominf$ = Promedio de valores de los píxeles menores al pixel central.

Y a partir de las ecuaciones (3.2) y (3.3) obtenemos el valor para el contraste:

$$C_{PC} = promsup - prominf \tag{3.4}$$

Donde:

C_{PC} = Valor de contraste del pixel central.

Al momento de realizar el cálculo del contraste, podemos encontrar que el valor de $numinf$ puede ser cero, lo cual sucede si todos los píxeles vecinos son mayores o iguales al pixel central. Igualmente, el valor de $numsup$ puede ser cero, si todos los píxeles vecinos son menores al pixel central. En el primer caso tomamos el valor del pixel central y se lo asignamos a $prominf$. Para el segundo caso, también tomamos el valor del pixel central, pero ahora se lo asignamos a $promsup$.

Con las fórmulas anteriores obtenemos los valores de brillo y contraste para los píxeles de la imágen analizada. A partir de estos valores podemos obtener los histogramas deseados. En la figura 3.2 aparecen unas imágenes representando cada una de las áreas que se están analizando, junto con sus respectivo histogramas. Hay que tomar en cuenta que en los histogramas, la parte de la izquierda corresponde al histograma de brillo, y la parte de la derecha al histograma de contraste.

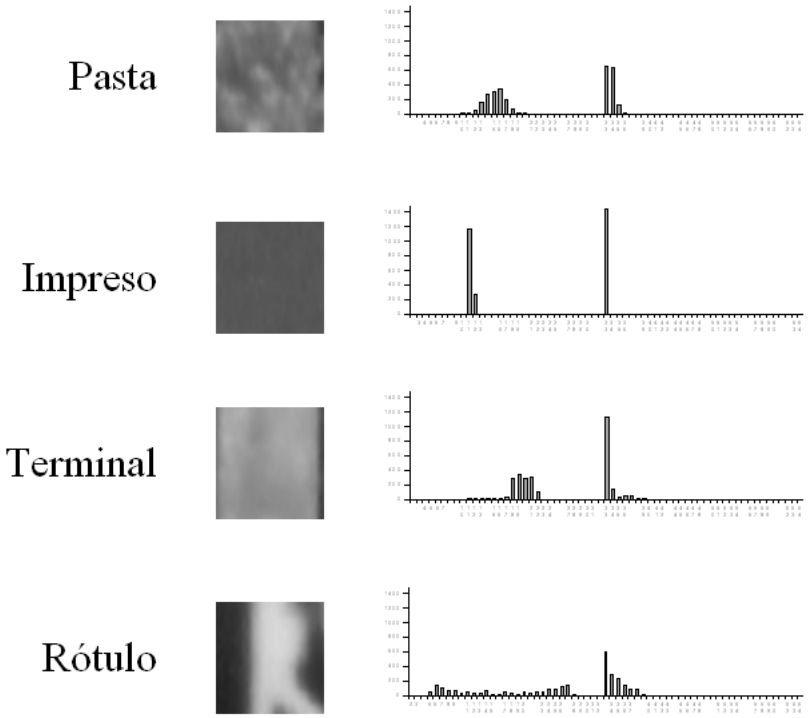


Figura 3.2: Histogramas de diferentes áreas de un circuito impreso.

Como se puede apreciar, cada área tiene unos histogramas que pueden ser diferenciados de los demás. Esto es cierto aún para las imágenes que son combinación de cualquiera

de las cuatro áreas mencionadas, por ejemplo, terminal con pasta o pasta con circuito impreso.

La siguiente sección, que trata acerca del algoritmo para comparación de imágenes, explica como son usados los histogramas de brillo y contraste para realizar tal comparación

3.3. Método propuesto

Por lo expuesto en el capítulo 2, el sistema de inspección está basado en comparación de imágenes. La propuesta fué la de usar un algoritmo que, haciendo uso de una imagen base y una de prueba, nos permita obtener un solo valor, el cual indicará el nivel de semejanza entre las imágenes. El uso de los histogramas explicados en la sección anterior nos permite obtener este valor. Para nuestro trabajo, usamos la técnica de intersección de histogramas [13] para obtener el nivel de semejanza.

Para explicar el algoritmo, consideremos primero dos histogramas hipotéticos, correspondientes al de la imagen base y una imagen de prueba, llamada Prueba 1, mostrados en la figura 3.3. Solo tomamos un histograma para este ejemplo, posteriormente se explicará como fueron usados los dos histogramas.

Hay que tomar en cuenta que estos histogramas se obtuvieron al analizar los datos de una imágen, por lo que el total de píxeles, 252 en este caso, corresponde al tamaño de la misma. Ahora, para intervalos correspondientes de los histogramas tomamos el que tiene el menor número de píxeles. Por ejemplo, para el intervalo con valores de 32 a 47, en el histograma base tenemos 8 píxeles y en el histograma de Prueba 1 tenemos 10, por lo que tomamos el valor de 8. De esta forma procedemos con los demás intervalos. Como resultado, formamos otro histograma, el cual es mostrado en la figura 3.4.

Para nuestro análisis, el punto importante a notar es el número de píxeles que aparecen en este último histograma, 241 en este caso. A este número le llamaremos suma de valores mínimos y es el que nos indica la semejanza entre las imágenes. Entre más parecidas sean dos imágenes, el número de píxeles se aproximará al total de píxeles de la imagen base

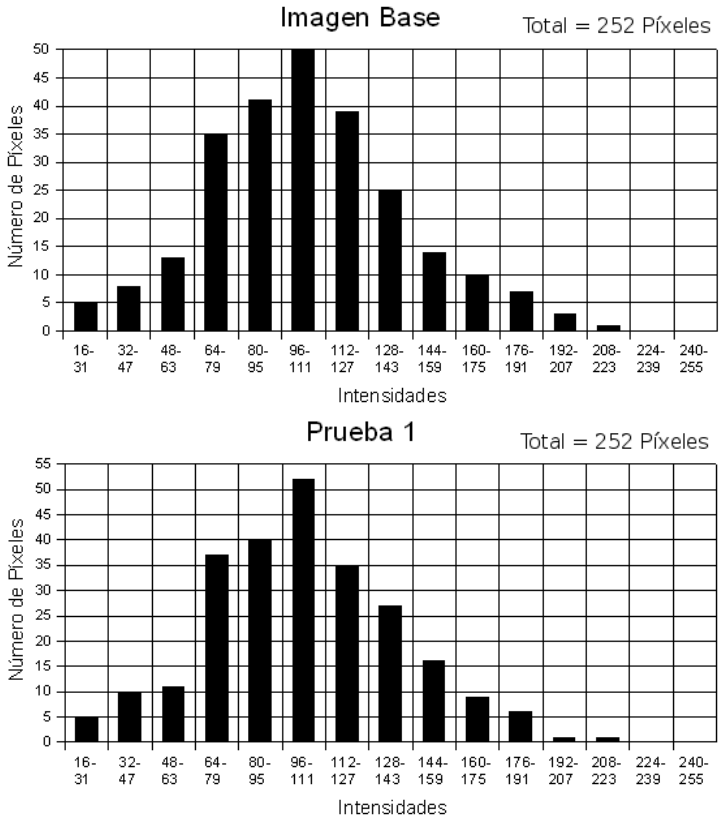


Figura 3.3: Histogramas de la imagen base y de imagen de prueba 1.

(o de las imágenes de prueba, ya que ambas tienen el mismo número de píxeles). Por el contrario, si dos imágenes son distintas, el número será menor. Esto último lo podemos ver al analizar el histograma mostrado en la figura 3.5, llamado Prueba 2, el cual es distinto al histograma de la imagen base.

En la figura 3.5 se puede observar que la mayor cantidad de píxeles se encuentra más hacia la derecha. Sin embargo, el total de píxeles en la imagenes de prueba deben de ser iguales al de la imagen base. Como en el caso anterior, para cada intervalo de los histogramas base y Prueba 2, tomamos el valor mínimo. Con esto formamos otro histograma, mostrado en la figura 3.6. A diferencia del histograma de la figura 3.4, en donde la forma era parecida al del histograma base, ahora aparece un histograma distinto. Pero lo más importante a notar es el número de píxeles, 94. Como se puede apreciar este es un valor mucho menor al del caso anterior, indicando que los histogramas corresponden a imágenes

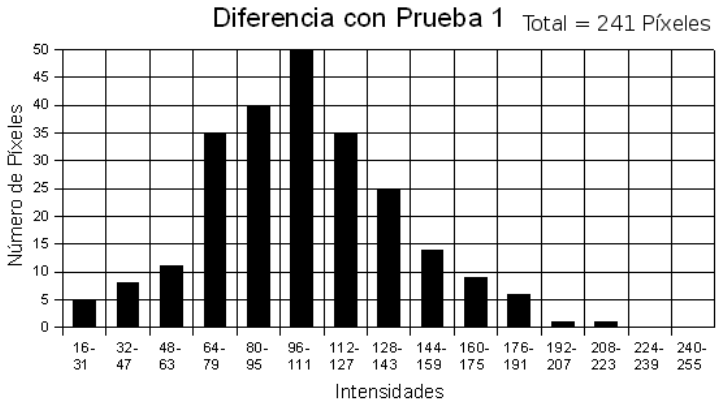


Figura 3.4: Resultado de tomar los valores mínimos de cada intervalo.

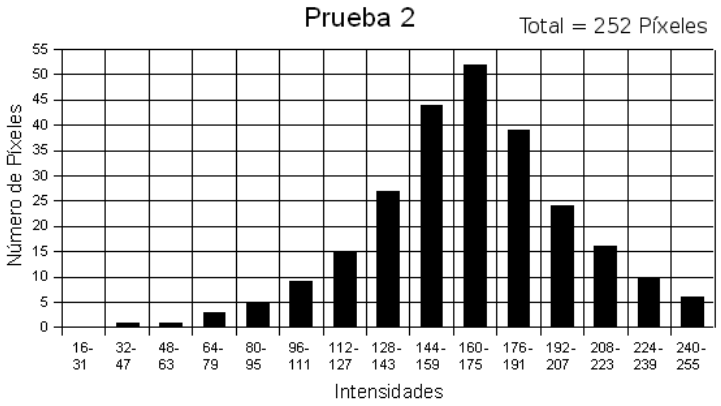


Figura 3.5: Histograma correspondiente a una seguna imagen de prueba.

distintas.

Lo anterior nos indica que debemos seleccionar un número de píxeles N_u, el cual sirve como umbral. Si la comparación de histogramas genera una suma de valores mínimos mayor o igual a este umbral, entonces las imágenes correspondientes se consideran iguales. Pero si es menor al umbral, las imágenes son distintas. Puesto que este algoritmo será usado para el sistema de inspección de tarjetas, el decir que dos imágenes son iguales equivale a decir que en la imagen de prueba no hay errores en el depósito de la soldadura. Dos imágenes distintas indican que en la tarjeta de circuito impreso, la que corresponde a la imagen de prueba, existe una falla en el depósito de soldadura.

Como se explica en el artículo de Swain[13], se usan histogramas normalizados para realizar la comparación. Sin embargo, con esto la suma de los valores mínimos de los

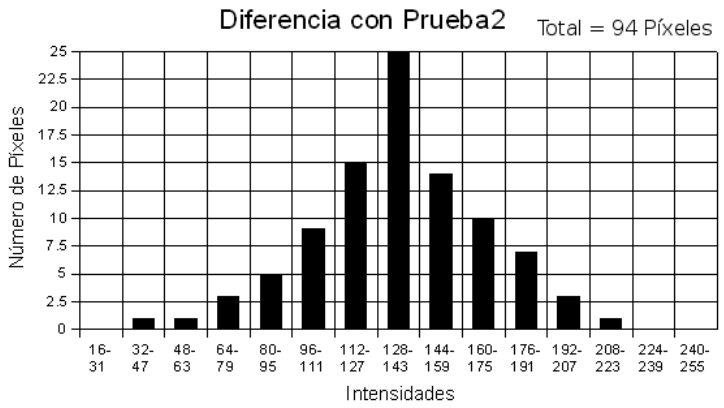


Figura 3.6: Valores mínimos entre los histogramas de la imagen base y prueba 2.

histogramas varía de 0 a 1; mientras más se acerce a 1 esta suma, las imágenes son semejantes. Aquí decidimos no usar histogramas normalizados, porque ello implicaría el uso de números con punto decimal. Puesto que el proyecto que se está desarrollando será implementado en FPGA, el uso de números con punto decimal obligaría a usar una mayor cantidad de recursos, y por lo tanto de más espacio del circuito. Por ello, el algoritmo explicado hace uso de números enteros.

Para el presente proyecto, el tamaño de las imágenes usadas es de 640x480 píxeles, ya que esta es la resolución del dispositivo sensor de imagen. Con este tamaño, no es posible aplicar el algoritmo a la imagen completa, porque existen variaciones pequeñas con respecto al tamaño de la imagen que es necesario detectar. Sin embargo, se deben permitir variaciones tolerables entre las imágenes. Para explicar mejor esto, consideremos las imagenes mostradas en la figura 3.7, La primera imagen es la que se toma como Base. La segunda imagen, llamada Prueba1, es un poco distinta a la imagen original, pero el sistema la debe de clasificar como igual. La tercera imagen, llamada Prueba 2, presenta unas variaciónes con respecto a la imagen base, y debe ser detectada como distinta porque se tratan de fallas de depósito.

Si aplicamos el algoritmo a las imagenes, y ajustamos el nivel de umbral a un valor muy elevado, el programa detectaría erroneamente distintas a las imágenes Base y Prueba1 (que deben de ser clasificadas como iguales), lo cual es debido a las pequeñas variaciones

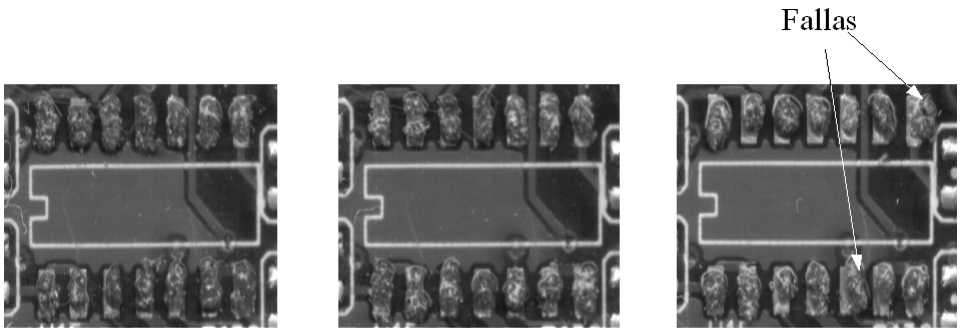


Figura 3.7: Comparación entre tres imágenes

entre ambas imágenes. Si el umbral es muy bajo, el programa sería incapaz de detectar variaciones en las tres imágenes, clasificando de manera erronea a la imagen Base y Prueba 2 como iguales. Después de realizar varios intentos, no fué posible encontrar un valor del umbral adecuado para distinguir las imágenes. Para eliminar este problema, en lugar de analizar la imagen completa, segmentamos las imágen en cuadros disjuntos, y comparamos sucesivamente segmentos respectivos de las imágenes de prueba y referencia, como se muestra en la figura 3.8.

El tamaño de los segmentos se obtiene de manera experimental, y depende del tamaño de detalle que queremos analizar. Si es muy pequeño, el programa marcará como distintas dos imágenes con variaciones muy pequeñas. Si es muy grande, será dificil detectar los detalles deseados. Para nuestro caso, en donde manejamos imagenes de 640x480 píxeles, encontramos que un valor de segmento de 40x40 píxeles es el adecuado, con ello logramos encontrar las diferencias que queríamos detectar, permitiendo variaciones pequeñas tolerables de las imágenes.

Para realizar la comparación de las imágenes, obtenemos el histograma de brillo y contraste de cada uno de los segmentos correspondientes. Por ejemplo, y como está indicado en la figura 3.8, tomaríamos el segmento A1 de la imagen base y de la imagen de prueba. Posteriormente, se realiza la comparación entre segmentos correspondientes, buscando alguna diferencia.

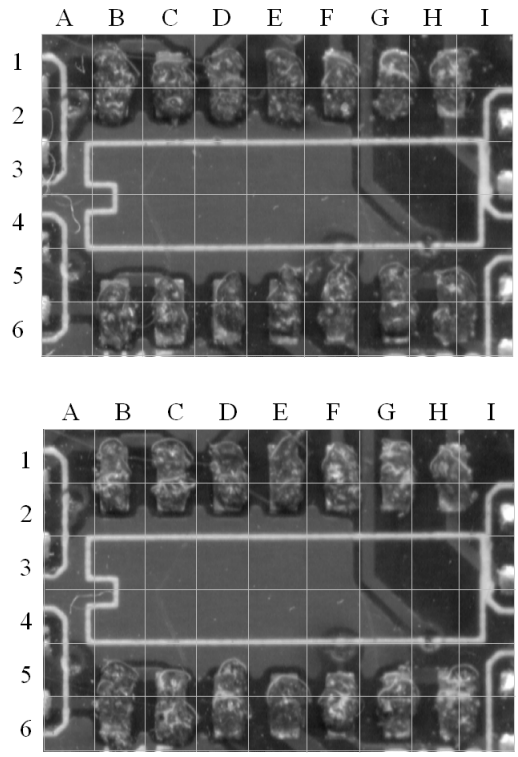


Figura 3.8: Comparación de segmentos correspondientes.

Ahora, para formar los histogramas, dividimos todo el rango de intensidades de los píxeles entre 32 intervalos. Puesto que formaremos dos histogramas, tendremos en total 64 valores.

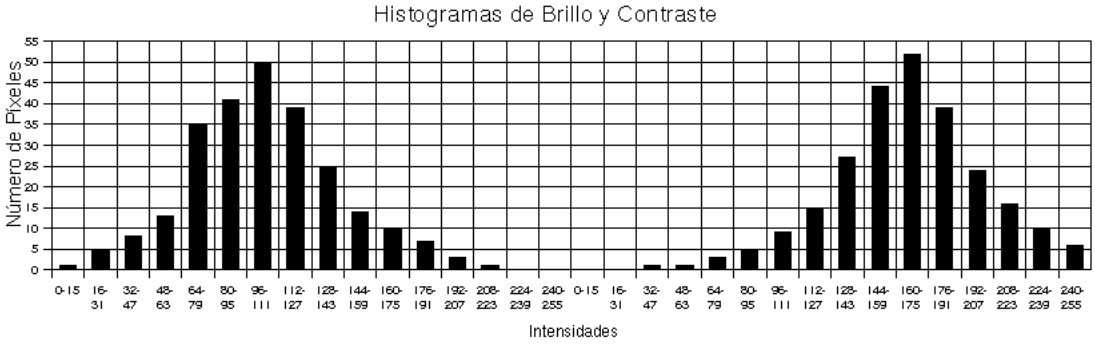


Figura 3.9: Unión de los histogramas de brillo y contraste.

Con el método descrito anteriormente, obtenemos los histogramas de los valores mínimos para el brillo y el contraste. Esto es mostrado en la figura 3.9, en donde solo se muestran 16 intervalos para cada histograma. Después de realizar la suma de píxeles para

cada uno de los histogramas de valores mínimos, aplicamos un peso distinto para cada suma. Esto es para eliminar las posibles fallas al analizar las imágenes debido al cambio de brillo. Para los experimentos realizados, encontramos que con unos valores del peso para el brillo P_B igual a 0.3, y del peso para el contraste P_C de 0.7, obteníamos los mejores resultados.

Finalmente, y de acuerdo a la técnica de intersección de histogramas [13], dos imágenes serán iguales si:

$$P_B * summin_{HB} + P_C * summin_{HC} > N_u \tag{3.5}$$

Donde:

$N_u = P_U * n$ Es el umbral que define la semejanza entre imágenes.

n = número total de píxeles analizados por cada segmento de imagen.

P_U = Porcentaje del total de píxeles usado para definir la semejanza, entre 0 y 100.

P_B , P_C = Pesos de los histogramas de brillo y contraste, respectivamente, entre 0 a 100, tal que $P_B + P_C = 100$.

$summin_{HB}$, $summin_{HC}$ = suma de valores mínimos de los histogramas de brillo y contraste, respectivamente.

El proceso se realiza de manera sucesiva entre los segmentos de imagen correspondientes. Si encontramos uno o mas segmentos distintos, entonces las imágenes son distintas. Puesto que las imágenes analizadas corresponderán a tarjetas con soldadura en pasta depósitada, el encontrar dos imágenes distintas, equivale a decir que la tarjeta de prueba tiene una falla en el depósito de soldadura.

Al momento de escribir esta tesis, aún no se contaba con la parte de captura de imágenes. Una vez terminado el sistema completo, las imágenes serán obtenidas a partir de un sistema controlado, en donde la variación en la posición de las tarjetas será mínimo. Además las condiciones de iluminación serán controladas, lo que garantiza que todas las imágenes tendrán la misma intensidad de luz. Para el desarrollo de la presente tesis,

algunos experimentos los realizamos utilizando una cámara digital, capturando imágenes con las mismas características de tarjetas circuito impreso con pasta depósitada. En estos experimentos encontramos que, variaciones muy pequeñas en las intensidad de la luz, provocaban que las imágenes se detectaran distintas. Para eliminar este problema, dimos diferentes pesos a la suma de mínimos de los histogramas, dando un valor menor al histograma de brillo. Es por ello que aparecen los parámetros P_B y P_C en la ecuación (3.5) que nos indica el nivel de semejanza entre las imágenes.

3.4. Uso de Java para probar el algoritmo

Antes de implementar al algoritmo propuesto en FPGA, primero fueron realizados varios experimentos para verificar su funcionalidad. Para ello utilizamos el lenguaje de programación Java, el cual ya tiene incorporadas funciones para leer archivos de imágenes, y para manejar en una matriz los valores individuales de cada pixel.

En la figura 3.10 aparece la ventana de la aplicación. En ella aparecen tres botones y dos áreas para cargar imágenes. Con la ayuda del boton etiquetado como Base, cargamos en el área de la izquierda la imagen base. De forma parecida, con el botón Prueba se carga en el área derecha la imágen de prueba. Una vez cargadas ambas imágenes, con el botón Analiza se inicia el proceso de comparación. Cuando el programa termina el análisis, puede ser que aparezcan en ambas imágenes uno o más cuadros (como los que se muestran en la figura 3.10), dependiendo si el programa encontró segmentos distintos entre las imágenes. Estos cuadros indican las áreas en donde se encontraron las diferencias. Si las imagenes son iguales, después de terminado el análisis las imágenes permanecen sin cambios.

El programa analiza sucesivamente segmentos de 40x40 píxeles, comenzando por la esquina superior izquierda y avanzando de izquierda a derecha, y de arriba hacia abajo. Por cada segmento, el programa llama a una función llamada *segmento*, que recibe los segmentos de la imagen base y de prueba, obtiene los histogramas y realiza la comparación con el método descrito en la sección anterior. La función devuelve un valor de 1 si los

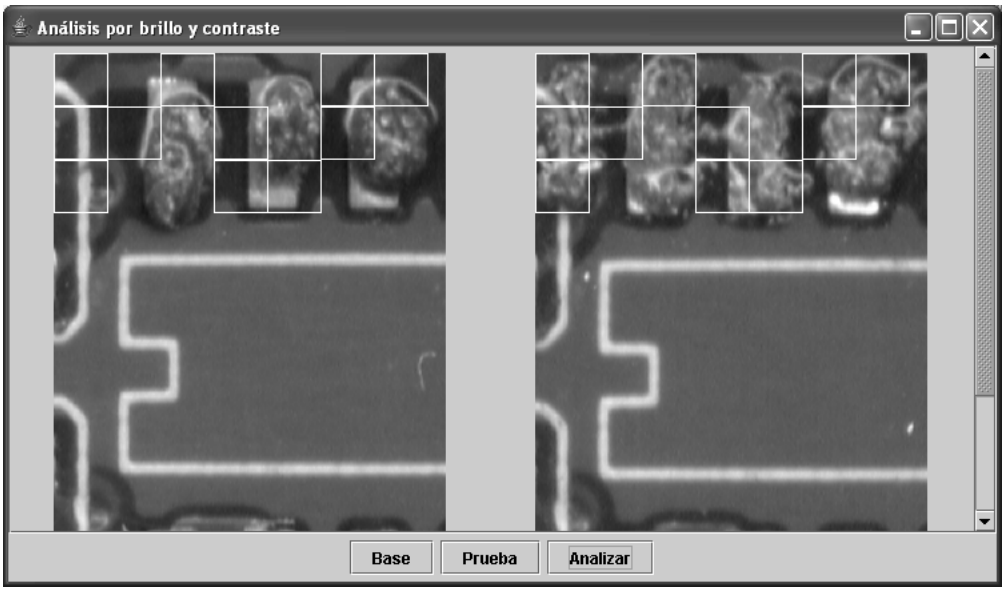


Figura 3.10: Ventana del programa en java que implementa el algoritmo.

segmentos son semejantes, y en caso que sean distintos devuelve un valor de cero. Cuando ocurre esto último, el programa llama a la función *cuadro*, que es usada para dibujar sobre ambas imágenes un cuadro, con el cual podemos ver en que parte de la imagen está la diferencia encontrada.

La forma como funciona el programa es parecida a su implementación en hardware. Esto es, la imagen será segmentada en cuadros de 40x40 píxeles y, a semejanza de la función *segmento* del programa de java, un circuito se encargará de realizar las comparaciones de los histogramas e indicar si los segmentos son semejantes. Pero, para reducir el tiempo necesario para enviar la información, son los valores de los histogramas de la imágen base los que serán enviados al circuito, en lugar del segmento de imagen. Con ello solo es necesario enviar 64 datos de 16 bits, contra los 1600 datos de 8 bits del segmento. Por ser ejecutado el programa en una computadora secuencial, la función *segmento* es usada de manera sucesiva para cada par de segmentos analizado. Por medio de un FPGA, podemos utilizar varios circuitos que realicen la misma función, todos ellos trabajando de forma paralela sobre distintos segmentos. Esto es lo que nos va a permitir reducir el tiempo necesario para la comparación de imágenes.

Antes de explicar la implementación del algoritmo en hardware, en el siguiente capítulo se da una breve introducción a los circuitos FPGA, así como las herramientas usadas para su programación.

Capítulo 4

Síntesis de FPGA's por medio de VHDL

Este capítulo tiene como fin el dar un panorama general sobre dispositivos programables, específicamente el FPGA. Se presenta la arquitectura del FPGA usado para el sistema desarrollado, además se explican las herramientas usadas para obtener el FPGA programado y funcionando.

4.1. Uso del FPGA para procesamiento de imágenes

Como fué mencionado en la introducción, los algoritmos de procesamiento de imágenes utilizan mucho tiempo para ejecutarse, cuando son realizados de manera secuencial. Podemos ver la razón de esto si analizamos el algoritmo de comparación de imágenes propuesto, en donde tomamos una ventana de 3x3 píxeles para asignar un valor al pixel central. Esta ventana tiene que recorrer todos los píxeles de la imagen, y para cada pixel hay que realizar una serie de operaciones. Esto trae como consecuencia el tener que realizar gran número de cálculos, proporcional al tamaño de la imagen.

El uso de DSP puede disminuir el tiempo de ejecución, pero muchos recursos del procesador quedan desperdiciados. Además persiste el problema de realizar los cálculos de

manera secuencial.

Otra forma de reducir el tiempo de ejecución es realizar los cálculos de forma paralela. Esto es posible en nuestro caso, ya que los valores de brillo y contraste para cada pixel se pueden obtener de manera independiente de los demás. Debido a que las computadoras convencionales solo tienen un procesador, no es posible tener un programa que ejecute el algoritmo propuesto de forma paralela. Esto no se aplica al caso de computadoras con múltiples procesadores. Sin embargo, su uso implica un gran costo, por lo que no es posible usarlas en este proyecto, cuyo objetivo es obtener un sistema sencillo y económico.

Por otra parte, el uso de un FPGA nos permite obtener el paralelismo deseado, ya que en este circuito es posible tener físicamente varios procesadores. Además se elimina el problema de los costos, ya que es mucho más económico desarrollar un prototipo funcional en FPGA, comparado con el uso de una computadora paralela.

En una situación ideal, podemos tener un procesador para obtener los valores de cada pixel. Pero como esta situación no es práctica, es preferible tener un procesador para diferentes segmentos de la imagen a analizar. Nuevamente, esto último se ajusta al algoritmo de análisis propuesto en el capítulo 3, en donde las imágenes son comparadas segmento por segmento en la búsqueda de diferencias. Así, cada uno de los procesadores se encargará de analizar un segmento de 40x40 píxeles, indicando por medio de una señal si los segmentos son iguales.

Como se puede apreciar, el uso de FPGA's para procesar imágenes nos da algunas ventajas. El propósito de esta tesis es determinar estas ventajas en cuanto a tiempo y costos. Por una parte se desea saber si el sistema cumple con los requerimientos de tiempo que impone la industria. Además, se desea estimar el espacio de FPGA necesario para implementar el procesamiento de imagen, ya que el sistema será mas económico mientras sea menor el espacio utilizado.

4.2. El arreglo de compuertas programable

Para el presente proyecto se usó un FPGA de la serie Spartan-III de Xilinx. Esta familia presenta varias características que la hacen apropiada para análisis de imágenes, como son multiplicadores y bloques de memoria. En la figura 4.1 se muestra el diagrama a bloques para los dispositivos de la familia Spartan-III.

La arquitectura de los FPGA de esta familia incluye 5 elementos principales [17]:

- Bloques lógicos configurables (CLB, Configurable Logic Blocks). Es el principal recurso para implementar circuitos lógicos, tanto síncronos como combinacionales. Cada CLB está formado por dos pares de slices. Cada slice contiene un generador de funciones lógicas, también conocido como tabla de consulta (LUT, Look-Up Table), y que pueden ser usados como generadores de funciones o memorias. Además contienen elementos para almacenamiento, compuertas aritméticas, múltiplexores, entre otros.

- Bloques de Entrada/Salida (IOB, Input/Output Blocks). Controlan el flujo de datos entre las terminales y los elementos internos del FPGA. Cada IOB soporta flujo de datos bidireccional más una operación de tres estados.

- Bloques de RAM. Permiten almacenamiento de datos en forma de bloques de 18 Kbits de puerto dual. Estos bloques nos permiten instanciar una RAM dentro del FPGA, lo cual es importante para nuestro proyecto, ya que es necesario almacenar los valores de los píxeles del segmento analizado.

- Multiplicadores. Estos bloques aceptan como entrada dos números de 18 bits, la salida es un número de 32 bits que representa el producto de los números de la entrada.

- Manejador de Reloj Digital(DCM, Digital Clock Manager). Los dispositivos de la familia Spartan III pueden tener dos o cuatro DCM's. Son usados como sintetizadores

de frecuencia para distribuir, multiplicar, dividir y desplazar en fase las señales de reloj.

Esta información es solo una breve descripción de las funciones de cada elemento. Para conocer más a detalle su funcionamiento, se recomienda consultar el manual de funcionamiento [16].

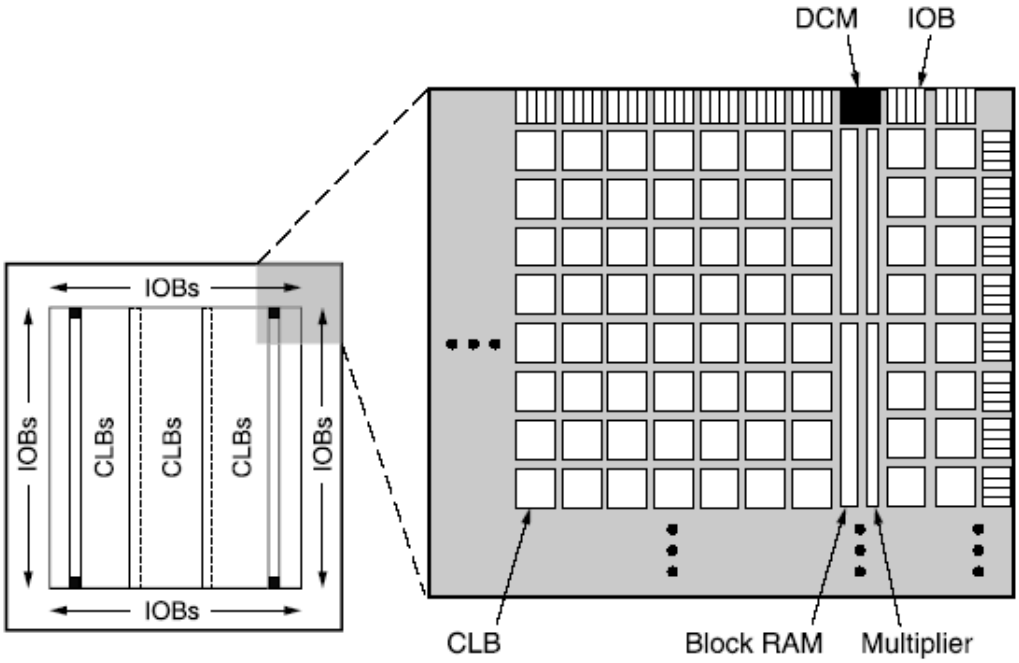


Figura 4.1: Diagrama de bloques basico para la familia Spartan III.

Como se puede apreciar en la figura 4.1, hay un anillo de IOB´s alrededor de un arreglo de CLBs. Además existen uno, dos o cuatro columnas de bloques de RAM, dependiendo del dispositivo. Cada bloque de RAM está asociado con un multiplicador. Los DCM están colocados en el extremo del bloque externo de la columna correspondiente.

Los FPGA's de la familia Spartan son de tipo SRAM, esto quiere decir que utilizan una memoria RAM cuyas celdas controlan los elementos y la conexión entre ellos. Se le llama configuración al proceso de programar esta RAM para que el FPGA realice la función que deseamos. De esta forma, podemos hacer que el FPGA funcione como un microprocesador, como una RAM, o cualquier otro dispositivo lógico. Los datos para la configuración del FPGA deben ser almacenados en algún dispositivo externo no volatil, como una PROM,

esto es debido a que la SRAM pierde su información al estar el dispositivo apagado. Al momento de aplicar el voltaje de alimentación los datos son escritos en el FPGA, y después de esto tenemos un circuito funcional.

Esta fué una breve explicación sobre las características del FPGA y su utilidad para implementar al algoritmo de comparación de imágenes. En las siguientes secciones se explican las herramientas usadas para obtener un FPGA programado.

4.3. El lenguaje de descripción VHDL

Como se mencionó en la sección anterior, el FPGA usado es de tipo SRAM, lo que quiere decir que debe ser programado cada vez que se alimenta con energía y pueda realizar la función deseada. Para realizar la programación a la que se hace mención, es necesario tener un archivo con la información a ser cargada en la RAM del FPGA, denominado archivo de configuración o *bitstream*. Para obtener este archivo se hace uso de lenguajes de descripción de hardware, junto con los compiladores correspondientes. En esta sección se da una breve explicación de los lenguajes de decripción de hardware, en la siguiente se explican las herramientas usadas para obtener el archivo de configuración (*bitstream*)

Los lenguajes de descripción de hardware consisten de una serie de instrucciones y directivas, parecidas a las usadas en los lenguajes de programación, como C o Java. Pero a diferencia de los lenguajes de programación, los lenguajes de descripción de hardware no se usan para implementar un algoritmo en una computadora. Como su nombre lo implica, son usados para describir componentes que forman parte de un circuito, y su interconección.

Dos son las diferencias fundamentales entre los lenguajes de descripción de hardware y los lenguajes de programación [15]. La primera es que en los lenguajes de descripción las instrucciones no son ejecutadas necesariamente de manera secuencial, sino que las salidas de los componentes descritos pueden cambiar su valor simultáneamente. La segunda diferencia es que los lenguajes de descripción deben de soportar nociones de tiempo real,

esto es con el fin de poder estimar el tiempo que tarda un circuito completo en dar el resultado esperado.

Para el presente proyecto, el lenguaje de descripción usado es VHDL. Con la ayuda de este lenguaje se puede crear el código necesario tanto para la simulación (probar la funcionalidad de un circuito), como la síntesis (creación de una lista de componentes básicos y sus conexiones). VHDL contiene elementos que son parecidos a los encontrados en lenguajes de programación, sin embargo hay que tener presente el hardware que generan al momento de ser usados. A continuación se mencionan algunos de los elementos comunes que encontramos entre VHDL y los lenguajes de programación:

- Uso de variables y constantes, para almacenar valores que pueden cambiar o valores que no cambian durante la ejecución del programa, respectivamente.

- Funciones y procedimientos, que son segmentos de código que pueden ser llamados desde distintos puntos. Al igual que en los lenguajes de programación, las funciones devuelven un valor y los procedimientos no. La diferencia es que son usadas para generar circuitos lógicos. En el caso de las funciones generan lógica combinatoria. Los procedimientos son usados dentro de los procesos, que serán explicados más adelante.

- La sentencia *case*, usadas para ejecutar una acción de acuerdo al valor de una variable. Son muy utilizadas para generar multiplexores o máquinas de estados finitos.

- La sentencia *if-then-else*, que realiza una prueba para determinar la veracidad o falsedad de una sentencia, y de acuerdo a esto realizar o no un grupo de instrucciones.

- Uso de ciclos *for, do* y *while*, ejecuta un grupo de instricciones hasta que se cumpla alguna condición.

- Uso de bibliotecas y paquetes, en donde se puede almacenar las definiciones de componentes, funciones, procedimientos o constantes, con la finalidad de tener compo-

nentes reutilizables, y así usar el mismo código para mas de un proyecto.

- Instrucciones de entrada y salida, usadas solo durante la simulación. Con este tipo de instrucciones es posible leer datos de un archivo, los cuales pueden ser los valores de una señal de entrada. También es posible generar archivos de salida, o enviar mensajes en la pantalla.

Algunos de los elementos mencionados son propios para la simulación y otros son tanto para simulación como para síntesis. Por ejemplo, la instrucción *wait for XXns* la cual es usada para esperar cierto intervalo de tiempo. Esta instrucción es muy usada en simulación, para variar el valor de señales por ejemplo, pero en síntesis no tiene sentido ya que sus uso implicaría conectar en cascada gran cantidad de elementos, para generar el retardo especificado. Otras instrucciones son ignoradas al momento de realizar la síntesis, por ejemplo la inicialización de variables y las instrucciones de entrada y salida.

En el lenguaje VHDL encontramos instrucciones para el manejo del tiempo, como lo es la instrucción *wait* mencionada anteriormente. También es posible suspender la ejecución del programa hasta que ocurra cierto evento.

Además de las variables y constantes, en VHDL aparece el concepto de señal. Éstas pueden considerarse como segmentos de alambre que conectan la salida de un componente con la entrada de otro. Las variables son representaciones que se usan para la escritura del programa en VHDL, después de la síntesis generan ya sea registros o alambres, de acuerdo a su ubicación en el programa de VHDL.

Tanto para señales como para variables existe el concepto de tipo, usado para definir los valores que se les pueden asignar. Existen tipos de datos como bit, boolean, integer y character. Su uso es parecido al de los lenguajes de programación. Solo hay que tomar en cuenta cómo serán sintetizadas las señales o variables de acuerdo al tipo. Por ejemplo una señal tipo bit o boolean generan un alambre, pero una señal tipo integer con valores de 0 a 255, genera un bus de 8 lineas. Una variable de tipo integer con valores de 0 a 255 puede generar un registro de 8 bits.

Otro aspecto fundamental de VHDL es el concepto del par entidad(*entity*) - arquitectura(*architecture*), la cual es usado para representar cualquier componente del circuito diseñado, ya sea una compuerta, un sumador, un flip-flop, una memoria, e inclusive un microprocesador completo. La entidad describe la interfaz del componente, o sea, sus entradas y salidas, y que en el lenguaje VHDL son conocidos como puertos. La arquitectura describe la funcionalidad del componente, es decir, la forma en que utiliza las señales de entrada para generar las señales de salida.

Otra característica de VHDL, aunque no es propia de los lenguajes de descripción, es que se puede establecer una jerarquía entre los componentes que forman un diseño. Por ejemplo, un sumador completo puede estar formado de dos sumadores simple más una compuerta OR, como se muestra en la figura 4.2. A su vez, es posible formar el sumador simple a partir de varias compuertas lógicas. Finalmente, las compuertas son las entidades más elementales, a partir de ellas obtenemos los demás componentes. Es esta jerarquía la que nos permite manejar diferentes niveles de abstracción, permitiéndonos concentrarnos en las funciones que deben realizar los componentes de jerarquías superiores, y posteriormente definir la estructura mas detalladamente, conforme bajamos en la estructura jerárquica de los componentes.

Para explicar mejor la jerarquia de componentes, así como el uso de entidades y arquitecturas para definirlos, tomemos como ejemplo un sumador completo, mostrado en la figura 4.2. En la parte superior de la jerarquia se encuentra el sumador completo, con sus entradas A, B y CEN, y sus dos salidas SALIDA y ACARREO.

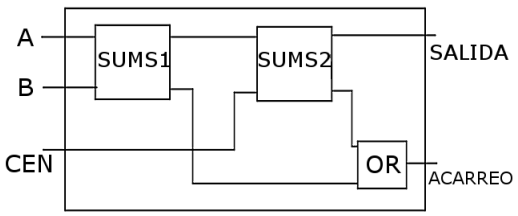


Figura 4.2: Un sumador completo a partir de dos sumadores simples y una compuerta.

A su vez, el sumador completo está formado por dos sumadores simples SUMS1 y

SUMS2, y una compuerta OR, cada uno con sus respectivas entradas y salidas. Con VHDL, primero definimos los sumadores simples y la compuerta OR con la ayuda de sus respectivas entidades y arquitecturas. Puesto que los sumadores simples son iguales, podemos definir solamente uno, y posteriormente usar esta defición dos veces para tener los dos sumadores. En el listado de la figura 4.3 aparece el código para el sumador simple, y como se puede ver usamos los operadores de VHDL *xor* y *and* para definir la funciónalidad del sumador simple. Esto nos permite ver la asignación concurrente a las salidas SUMA y ACARREO, ya que si cambia el valor de A, B o ambas, las dos salidas cambian de valor en el mismo instante de tiempo.

```
entity SUMSIM is
port (A,B : in bit;
      SUMA,ACARREO : out bit
      );
end SUMSIM

architecture  CONCURRENTE of SUMSIM is
begin
    SUMA <= A xor B;
    ACARREO <= A and B;
end BEHAVE;
```

Figura 4.3: Definición del sumador simple.

En la figura 4.4 está la definición de la compuerta OR usada en el sumador completo. Este componente no es necesario en un diseño real, ya que VHDL incluye un operador *or* para realizar esta función. La definición de este componente es para mostrar el uso de los procesos en VHDL. Estos son bloques de código que pueden estar solamente dentro de una arquitectura, y cuya característica principal es que las instrucciones que contienen se ejecutan de manera secuencial, parecida a los lenguajes de programación. Dentro de una arquitectura se pueden combinar sentencias concurrentes y procesos. Además puede existir más de un proceso en una arquitectura, en este caso cada proceso se considera como una sentencia concurrente, por lo que todos los procesos de una arquitectura se ejecutan en paralelo.

```
entity OR is
port (A,B : in bit;
      SALIDA : out bit
     );
end OR;

architecture SECUENCIAL of OR is
begin
COMP_OR: process(A,B)
begin
   if(A='1' or B= '1') then
      SALIDA <= '1';
   else
      SALIDA <= '0';
   end if;
end process COMP_OR;
```

Figura 4.4: Definición de una compuerta OR.

Finalmente en la figura 4.5 aparece el código de VHDL para el sumador completo. Aquí hay una diferencia con respecto a los códigos para el sumador simple y la compuerta OR. Para estos casos se utilizó un estilo de programación llamado comportamental, ya que en la arquitectura fué definida la función que realizaban los componentes. Para el caso del sumador completo aparece otra forma de escribir el código de VHDL, llamado estructural. Como se puede ver en la figura 4.5, se instancian el sumador simple y la compuerta definidos anteriormente por medio de la palabra clave *component*. Después, estos componentes son usados en la arquitectura para definir las interconexiones entre los mismos y los puertos de la entidad superior, en este caso el sumador completo. Este código también nos permite ver el uso de las señales que, como se mencionó anteriormente, pueden considerarse como alambres para conectar las salidas con las entradas entre los componentes. El circuito generado es el que se muestra en la figura 4.2.

Esta es solo una muy breve explicación del lenguaje VHDL, solo usado para mostrar las características que posee y el porqué es útil en la descripción de circuitos. Para una introducción más detallada se requiere consultar libros de texto especializados en el tema [18] [4] [5].

```
entity SUMCOM is
port (A,B, CIN : in bit;
      SALIDA, ACARREO : out bit
     );
end SUMCOM;

architecture STRUCT of SUMCOM is
signal S1, S2, S3 : bit;
component SUMSIM
   port(A,B : in bit;
        SUMA,ACARREO : out bit
       );
end component;
component OR
   port(A,B : in bit);
        SALIDA : out bit
       );
end component;

begin
   SUMS1:SUMSIM port map(A,B,S1,S2);
   SUMS2:SUMSIM port map(S1,CIN,SALIDA,S3);
   OR:OR port map(S3,S2,ACARREO);
end STRUCT;
```

Figura 4.5: Definición de una compuerta OR.

4.4. Proceso para el diseño de FPGA

Para obtener un FPGA funcional, se debe pasar por una serie de etapas, las cuales son mostradas en la figura 4.6. Como se puede ver, hay un orden que debe ser seguido para obtener el diseño definitivo. En varias etapas del diseño pueden ser realizados varios tipos de simulación, esto es porque conforme se avanza en las etapas, es necesario considerar las restricciones que impone el circuito, como los retardos en los elementos. En cualquier caso, si la funcionalidad del diseño no es la esperada, como resultado de realizar la simulación, es necesario revisar el código de VHDL para conseguir la funcionalidad requerida. En la figura aparecen en un recuadro tanto las etapas de mapeo como la de colocación y ruteo, esto es porque a estas etapas en conjunto se les conoce como implementación. A

continuación se da una explicación de cada una de las etapas.

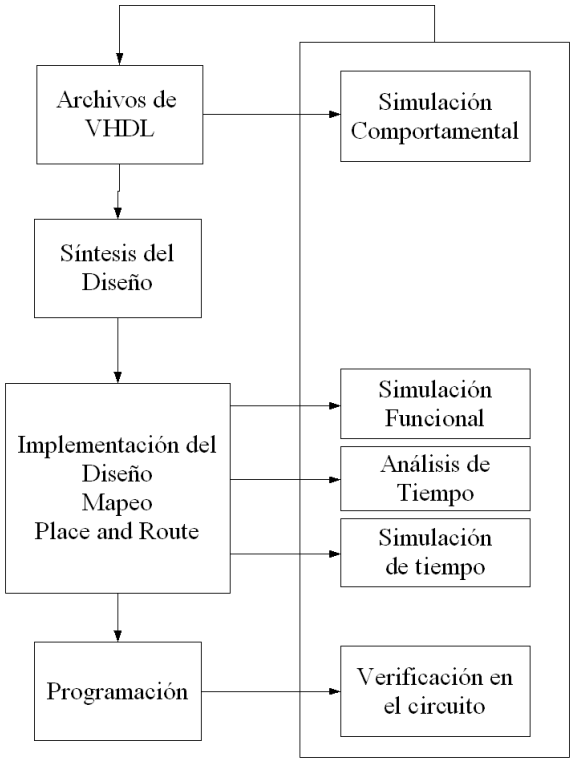


Figura 4.6: Etapas para el diseño en FPGA.

4.4.1. Creación de archivos de diseño en VHDL

En esta parte se genera el programa en VHDL, el cual es un archivo de texto con la descripción de hardware deseada. En este archivo se describen los diferentes componentes que forman el circuito diseñado. El texto completo puede quedar en un solo archivo, o puede ser dividido en varios segmentos para tener archivos más compactos. Si se hace esto último, hay que tener en cuenta que VHDL define las unidades mínimas que deben estar en un archivo, llamadas unidades de diseño. Un solo archivo puede tener una o mas de estas unidades de diseño, y son las que se enumeran a continuación:

1. Entidad.

2. Arquitectura.

3. Configuración.

4. Declaración de un paquete.

5. Cuerpo del paquete

Los archivos de diseño pueden ser realizados con cualquier procesador de texto, aunque existen herramientas para simulación o síntesis que contiene su propio editor de texto.

4.4.2. Simulación comportamental del diseño

Una vez terminado la descripción de VHDL, es importante saber si el diseño así generado realizará la función deseada. Esto se puede lograr con la simulación del diseño, la cual además es útil para verificar la sintaxis del programa. Hay que tomar en cuenta que en este punto hace falta considerar algunos factores, como los retardos de tiempo inherentes de los componentes, por lo que esta simulación es solo una primera aproximación. En etapas posteriores se realizan otras simulaciones que ya consideran este tipo de factores.

4.4.3. Síntesis del diseño

En esta parte se crea una descripción a nivel de compuertas del diseño obtenido. El resultado de esta etapa es la obtención de una lista con las compuertas necesarias para implementar un diseño, y además de todas las conecciones necesarias entre las mismas. En esta etapa se puede realizar otra simulación, llamada funcional, para verificar que el arreglo de compuertas siga efectuando la función deseada.

4.4.4. Mapeo

El mapeo usa la descripción de compuertas generadas por la herramienta de síntesis, distribuyéndolas entre los diferentes recursos con los que cuenta el FPGA, como son los CLB's, bloques de memoria, multiplicadores, y otros.

4.4.5. Colocación y ruteo

En el mapeo fueron definidos los recusos del FPGA que se deben usar para implementar el diseño. Con la colocación se seleccionan los componentes específicos a ser usados, y con el ruteo se definen las conexiones entre los diferentes componentes seleccionados. Después de esta etapa puede ser realizada otra simulación, denominada temporal, cuyo propósito es asegurarse que el diseño obtenido cumpla con los requerimientos de tiempo. Es en esta etapa donde intervienen factores como retardo entre los elementos.

4.4.6. Programación

Después de haber cubierto todas las etapas anteriores, en este momento es posible obtener el archivo necesario para programar el FPGA, conocido como *bitstream*. La programación consiste en enviar este archivo al FPGA para que establezca las conexiones apropiadas entre los diferentes componentes del mismo.

4.5. Herramientas usadas para el diseño

La descripción que se acaba de dar del flujo de diseño para FPGA es general, cada fabricante peuede agregar una o más etapas al flujo. Sin embargo, las etapas mencionadas son las principales. Para este proyecto se utilizó el Ambiente de Software Integrado (ISE, Integrated Software Environment) de Xilinx, que consiste de un conjunto de herramientas usadas para cada una de las etapas del diseño descritas, más otras especializadas en tareas más específicas, como el analizador de tiempo, editor de restricciones, generador de núcleos (cores), entre otros. ISE tiene integrada una herramienta de síntesis, denominada XST. Sin embargo, para este proyecto, para realizar la síntesis fué utilizado Synplify Pro, ya que con esta herramienta logramos obtener mejor aprovechamiento del espacio de FPGA. La salida de Synplify Pro, un archivo de tipo EDIF, que puede ser usado con ISE para obtener el archivo de configuración, necesario para programar el FPGA.

Capítulo 5

Circuito básico para comparación de imágenes

El trabajo presentado en esta tesis forma parte de un proyecto cuya finalidad es el desarrollo de un sistema de inspección visual, para detectar fallas en el depósito de soldadura en pasta. De manera específica, para esta tesis se trabajó en la parte correspondiente al procesamiento de la imagen. Este capítulo presenta la arquitectura propuesta para llevar a cabo esta tarea, explicando a detalle el diseño del circuito básico para comparación de imágenes.

5.1. Arquitectura general del sistema

En la imágen 5.1 está representada la arquitectura del sistema de inspección propuesto. Cada una de las etapas es descrita más adelante dentro de esta sección. Hay que notar que en esta imágen solo se muestra la parte electrónica referente al procesamiento de la imágen. El sistema de inspección completo incluye otras etapas que serán desarrolladas en trabajos futuros. También es necesario mencionar que en esta tesis únicamente se desarrolló el circuito para implementar el bloque llamado "Analizador".

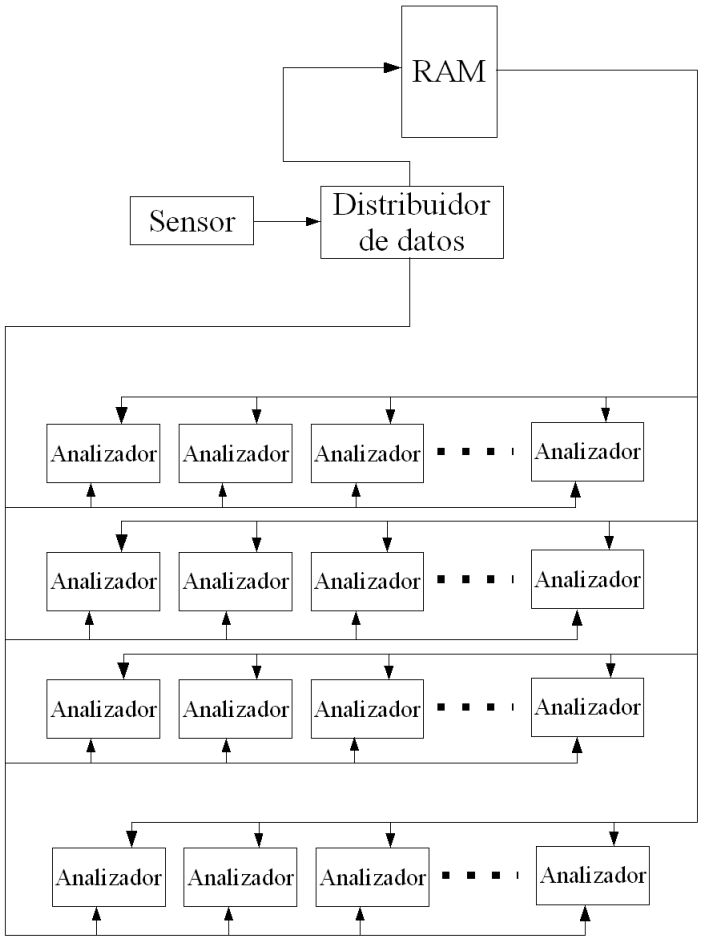


Figura 5.1: Arquitectura del sistema de inspección.

5.1.1. Captura de la imagen

El sistema de inspección contará con un dispositivo sensor de imágen, específicamente el circuito LM9627 de National Semiconductors. Se trata de un circuito CMOS usado para detectar imágenes en color. Sin embargo, la imagen generada por el sensor se encuentra en código Bayer, por lo que es necesario utilizar un algoritmo para convertir esta imagen en tonos de grises, formato usado por las siguientes etapas del sistema de inspección. El circuito sensor puede capturar imágenes con una resolución de 640x480 píxeles. Cada pixel está representado por un valor de 8 bits, que indica el nivel de color correspondiente a ese pixel, ya sea rojo, verde o azul. Después de realizar la conversión a tonos de grises, cada

pixel quedará representado igualmente por un valor de 8 bits, por lo que en la imagen se pueden manejar 256 diferentes tonos de grises, suficientes para realizar el análisis de textura.

5.1.2. Almacenamiento y distribución de datos

Los datos generados por el sensor, y convertidos a niveles de gris, son almacenados en una memoria RAM. Una vez dentro de esta memoria, esta etapa se encargará de distribuir segmentos de la imagen capturada entre las distintas unidades analizadoras. Previo a esta distribución, es necesario calcular los histogramas de brillo y contraste de cada uno de los segmentos de la imagen base, y almacenarlos en la misma memoria. Estos histogramas deben ser calculados solo una vez, cuando se presenta al sistema de inspección la tarjeta base para realizar las comparaciones, posteriormente son usados para analizar todas las tarjetas de prueba. De esta forma, cada unidad analizadora recibirá tanto los datos del segmento de imagen de prueba que debe analizar, así como de los respectivos histogramas de brillo y contraste de la imagen base.

5.1.3. Partición de la imagen

Podemos imaginar a la imagen capturada como una matriz de segmentos, como se muestra en la figura 5.2. Como fué explicado en el capítulo 3, cada segmento está formado por una matriz de N x N píxeles. Además se mencionó que el valor de N depende de los detalles que se desean analizar en la imagen, y su valor se obtiene experimentalmente.

Cada segmento pude ser analizado de forma independiente a los demás, lo cual es representado en la figura 5.2 donde existe una unidad básica de procesamiento (UBP) para cada segmento. Refiriéndonos ahora a la figura 5.1, cada una de las unidades analizadoras, correspondientes a las unidades básicas de procesamiento mencionadas anteriormente, reciben los datos almacenados en la RAM. Estos son los valores de los píxeles del segmento de imagen a analizar, más los datos del histograma de referencia. El envío de estos datos

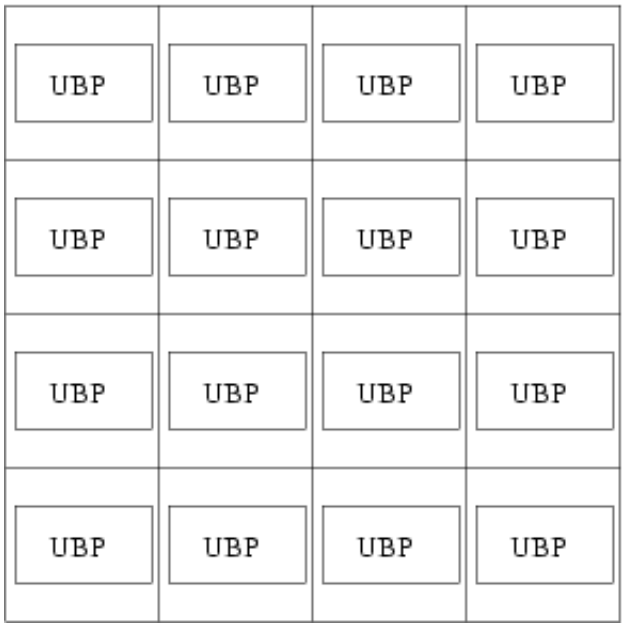


Figura 5.2: Partición de la imagen entre las unidades básicas de análisis de imagen.

está controlada por la unidad distribuidora, que genera las señales de direccionamiento adecuadas para la RAM, con el fin de recuperar los valores del segmento seleccionado. Esta misma unidad genera las señales para seleccionar la unidad analizadora a la que va dirigido el dato.

Como fué mencionado anteriormente, con el sensor usado tenemos imágenes con una resolución de 640x480 píxeles. Esta imagen es particionada en segmentos de 40x40 píxeles, por lo que se forma una matriz de segmentos de 16 columnas por 14 filas, que implica tener 224 segmentos. Sí tuviésemos una unidad analizadora por cada segmento, sería necesario utilizar la misma cantidad de unidades. El usar tal cantidad de circuitos podría traer como consecuencia una mala utilización del FPGA, porque el tiempo para analizar un segmento puede ser menor al tiempo de distribución de datos. La consecuencia de esto es que varias unidades analizadoras pueden quedar inactivas, antes de terminar la distribución de datos. Con el diseño de la unidad analizadora podemos conocer el tiempo que tarda en procesar un segmento de imagen, y con este dato poder determinar el número de unidades necesarias en el sistema, y así aprovechar al máximo el uso del FPGA.

5.1.4. Unidades analizadoras básicas de procesamiento

Estas unidades son las encargadas de procesar un segmento de la imágen, para obtener sus histogramas de brillo y contraste, y posteriormente realizar la comparación con los histogramas de referencia, indicando por medio de una señal si hay una diferencia entre los mismos. El desarrollo de estas unidades fué el trabajo principal para esta tesis, y su funcionamiento se explica en el resto de este capítulo.

5.2. Unidades básicas de análisis de imagen

Como fué mencionado al principio de este capítulo, la arquitectura presentada en la sección anterior será usada en un sistema de inspección visual. La finalidad de este sistema es tratar de reducir los tiempos de ejecución encontrados en los sistemas de inspección, así como los costos que implican. El desarrollo de las unidades analizadoras es importante, ya que con ellas podemos determinar el tiempo necesario para procesar una imagen completa.

La figura 5.3 muestra los bloques que constituyen las unidades básicas analizadoras. Cada una de ellas realizan las funciones que se indican a continuación:

1. **Entrada de datos**. La función de esta etapa es recibir, de manera ordenada, los datos provenientes de la unidad distribuidora. Esta etapa fué diseñada de manera provisional. Como prueba el circuito fué conectada al puerto paralelo de una computadora. Es por ello que aparecen las líneas marcadas como *ACK*, *LISTO* y *LEE*, las cuales son usadas para sincronizar la comunicación y evitar la pérdida de datos.

2. **RAM 16**. Esta es una memoria de 16 bits, con 128 localidades, usada para almacenar los datos correspondientes a los histogramas, tanto de la imagen base como la de prueba. Los datos de los histogramas de la imagen base, provenientes de la unidad analizadora, ocupan las primeras 64 localidades de esta memoria. Las localidades restantes son usadas para guardar los histogramas calculados del segmento recibido.

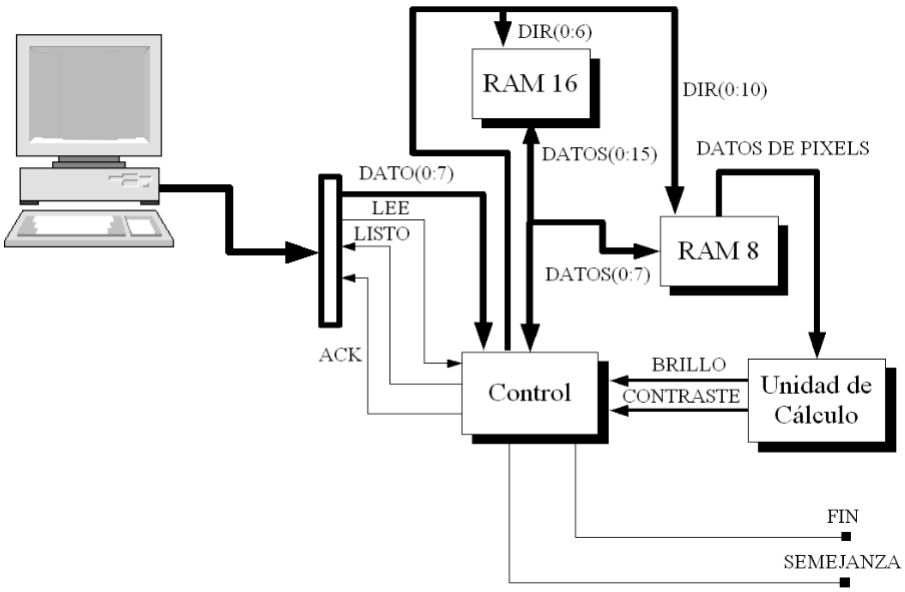


Figura 5.3: Arquitectura del circuito base.

3. **RAM 8**. En esta memoria de 8 bits se almacenan los datos correspondientes a los píxeles del segmento de la imagen a ser analizado. Como se manejan segmentos de 40x40 píxeles, esta memoria debe de tener 1600 localidades.

4. **Control**. Esta parte realiza tres funciones. Efectúa el proceso de comunicación con la unidad distribuidora para recibir los datos, y almacenarlos posteriormente en la RAM de 8 0 16 bits. Selecciona una ventana de 3x3 píxeles de la RAM de 8 bits, y con la ayuda de la unidad de cálculo obtiene los valores de brillo y contraste del pixel central, de esta forma puede generar el histograma. Finalmente, una vez obtenido el histograma del segmento de la imagen de prueba, lo compara con el recibido de la imagen base.

5. **Cálculo de brillo y contraste**. Recibe una ventana de 3x3 píxeles, para después calcular los valores de brillo y contraste correspondientes al pixel central.

Cada una de estas partes son explicadas más detalladamente en lo que resta de esta sección, así como la arquitectura usada para implementarla y así poder obtener el código de VHDL correspondiente. En las siguientes subsecciones está explicado más a detalle el

código usado para implementar la unidad de control, así como el usado para implementar algunas partes de la unidad analizadora. En el apéndice A se explica el funcionamiento de los demás componentes que forman la unidad analizadora, como el divisor, los comparadores o los registros, y que no fueron incluidos aquí con el fin de simplificar la explicación.

5.2.1. Entrada de datos

Esta etapa se encarga de recibir los datos desde el puerto paralelo de la computadora que envía la información. La forma como se reciben los datos es primero los valores para el segmento de 40x40 píxeles, seguido de los correspondientes a los histogramas, primero del histograma de brillo y después el de contraste. Para el caso de los histogramas, existen 32 intervalos para cada uno de ellos. Es por esto que la unidad analizadora recibirá 1600 valores del segmento de imagen, 32 para el histograma de brillo y 32 para el de contraste. Sin embargo, para cada intervalo del histograma son necesarios dos bytes para cada valor recibido. Esto indica que en total son 1600+64+64=1728 bytes que deben ser almacenados. Esto es representado en la figura 5.4. En ella se puede apreciar los datos que entran desde la izquierda. El primer dato, marcado como 0 y que corresponde al primer pixel del segmento de imagen, es almacenado en la localidad 0 de la RAM de 8 bits, el segundo dato es almacenado en la localidad 1, continuando este proceso hasta llegar al dato 1599, que corresponde al último pixel del segmento de imagen, y que es almacenado en la localidad 1599 de la memoria RAM de 8 bits. Hay que tomar en cuenta que la unidad analizadora espera recibir los píxeles del segmento fila por fila, empezando por la parte superior, como se muestra en la figura 5.5.

Después de los valores de los píxeles del segmento de imagen, aparecen los datos de los histogramas, los cuales deben ser almacenados en la memoria de 16 bits. Puesto que cada dato del histograma ocupa dos bytes, es necesario realizar dos lecturas por cada celda. Es por ello que en la figura 5.4 los datos 0 y 1 del histograma son almacenados en la parte baja y alta, respectivamente, de la localidad 0 de la RAM de 16 bits. Los primeros 32

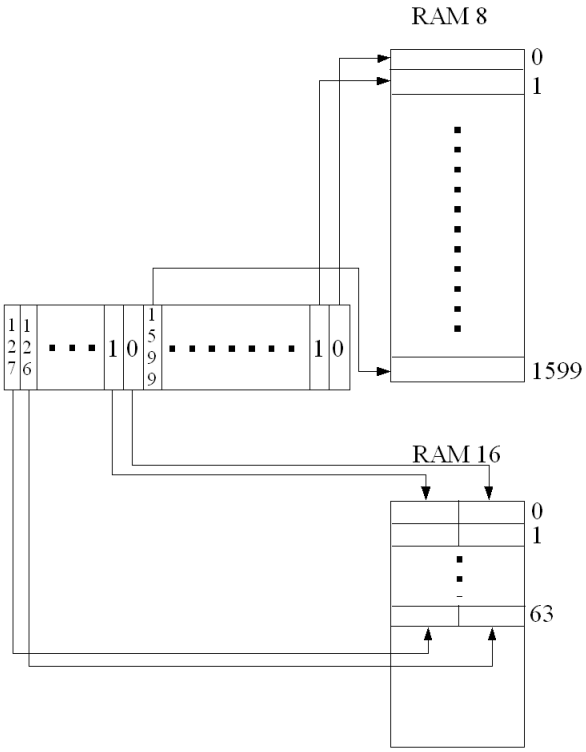


Figura 5.4: Distribución de datos entre las memorias.

datos (64 bytes) corresponden la histograma de brillo. Estos ocupan las localidades 0 a 31 de la RAM de 16 bits. Los últimos 64 bytes corresponden a los 32 datos del histograma de contraste, que son almacenados en las localidades 32 a 63 de la RAM.

Estos 64 datos recibidos ocupan la parte baja de la memoria de 16 bits, ya que la parte alta está reservada para almacenar los histogramas calculados del segmento recibido, y que son obtenidos en otra etapa.

Después de recibir estos 1728 bytes, la unidad analizadora comienza el cálculo de los histogramas. Durante este intervalo de tiempo, la unidad queda deshabilitada para recibir datos. Es hasta que termina el cálculo y comparación de histogramas cuando puede recibir otro bloque de datos para realizar otra comparación.

Como fué mencionado anteriormente, la etapa de entrada de datos se diseñó para que la unidad analizadora se conectará con el puerto paralelo de una computadora. Para conseguir la transferencia de información deseada, en la etapa de entrada se utiliza el protocolo usado para la comunicación con en el puerto paralelo. Esto fué hecho así porque,

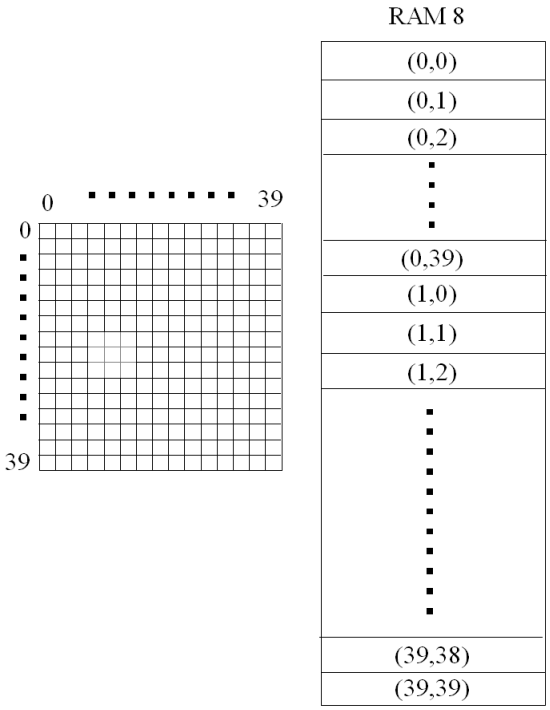


Figura 5.5: Distribución del segmento de imagen en la memoria de 8 bits.

para el desarrollo de este proyecto, solo se deseaba saber el tiempo que necesita la unidad analizadora para generar y comparar los histogramas. Queda para trabajo futuro definir la comunicación entre la unidad analizadora y la distribuidora de datos.

Para el caso de la unidad analizadora, los pasos usados para la recepción son los siguientes:

- Coloca en 1 la señal *LISTO*, indicando que se encuentrá lista para recibir información.
- Espera hasta que la señal *LEE*, proveniente de la unidad distribuidora, sea 1.
- Lee el dato correspondiente.
- Coloca en uno la línea de reconocimiento *ACK*.
- Espera hasta que la línea *LEE* sea 0.
- Coloca en 0 la línea de reconocimiento *ACK*.

Todos estos pasos son repetidos por cada dato recibido.

Por su parte, el programa que envía la información desde la computadora, debe de realizar los siguientes pasos:

- Espera en un ciclo hasta que la señal *LISTO* se encuentre en 1.
- Coloca el dato a ser leido.
- Pone en 1 la señal *LEE*, indicando a la unidad analizadora que el dato es válido y pude leerlo.
- Espera hasta que la línea de reconocimiento *ACK* este en 1.
- Coloca en 0 la señal *LEE*.
- Espera hasta que la línea de reconocimiento *ACK* este en 0.

Como en el caso anterior, todos estos pasos se repiten para cada dato enviado.

En la figura 5.6 está una primera parte del código VHDL usado para implementar el protocolo descrito, referente a la recepción de datos, y como se mencionó forma parte de la unidad de control. Aquí se puede ver que la señal *LISTO* es colocada en 1, indicando que la unidad puede recibir datos. En la figura aparecen otras señales, cuya función será explicada un poco más adelante. La señal *FIN*, sirve para indicar que la unidad analizadora terminó de procesar el segmento recibido, esto lo hace al colocar 1 por un ciclo de reloj al final del análisis. La unidad permanece en el estado 0 mientras la señal *LEE* (externa) sea 0. Cuando esta señal está en 1, comienza la recepción de los 1728 bytes.

Tan pronto como empieza la recepción de datos, la unidad de control pasa al estado 1, como se muestra en la figura 5.7. La escritura a las memorias RAM es deshabilitada, indicado por el 0 en las señales *HERAM8* y *HERAM16* de las memorias de 8 y 16 bits, respectivamente. Esto es con el fin de determinar primero a que memoria está destinado el dato recibido, así como la dirección de la localidad. La variable *SELRAM* es usada para

```
when 0 => LISTO <= '1';
        ACK <= '0';
        FIN <= '0';
        DIR <= "00000000000";
        HERAM8 <= '0';
        HERAM16 <= '0';
        DATMRAM <= "0000000000000000";
        SELRAM := '0';
        CONMUTA := '0';
        CONVIERTE <= '0';
        if LEE = '1' then
          ESTADO := 1;
          LISTO <= '0';
        end if;
```

Figura 5.6: Primera parte de la entrada de datos.

seleccionar la memoria adecuada. Cuando tiene un valor de 0, el dato recibido será almacenado en la RAM de 8 bits, y en caso contrario en la de 16 bits. Como primero son recibidos los datos del segmento de imagen, el valor de *SELRAM* es de 0, y el dato recibido (señal *DATO*) es asignada a la señal *DATMRAM*. Esta última es una señal de 16 bits, y está conectada a la entrada de ambas memorias. La memoria de 16 bits recibe todos los bits de *DATMRAM*, y la memoria de 8 recibe los primeros 8 bits (*DATMRAM(7:0)*), Como está seleccionada la memoria de 8 bits, los datos recibidos son almacenados en esta memoria.

```
when 1 => HERAM8 <= '0';
        HERAM16 <= '0';
        if SELRAM = '0' then
          DATMRAM(7 downto 0) <= DATO;
        else
          if CONMUTA = '1' then
            DATMRAM(15 downto 8) <= DATO;
          else
            DATMRAM(7 downto 0) <= DATO;
          end if;
        end if;
        ACK <= '1';
        ESTADO := 2;
```

Figura 5.7: Segunda parte de la entrada de datos.

Cuando se terminaron de recibir los datos del segmento de imagen, el valor de *SELRAM* es cambiado a 1 (este cambio es realizado en otro estado, explicado mas adelante). Puesto que deben recibirse dos datos de 8 bits para obtener un valor de 16 bits del histograma,

es necesario determinar que parte de la localidad de memoria de 16 bits almacenará el byte recibido. Esto se logra con la ayuda de la variable *CONMUTA*. Si esta variable tiene un valor de 1, el byte recibido va a la parte alta de la localidad de memoria direccionada, si es 0 será almacenada en la parte baja.

Sin importar la memoria seleccionada, y cumpliendo con el protocolo para recepción de datos, al final de este estado se coloca un 1 en la señal *ACK* para indicar que la unidad analizadora ya recibió el dato.

En la figura 5.8 se muestran la siguiente etapa de la recepción.

```
when 2 => if lee = '0' then
            ACK <= '0';
            ESTADO := 4;
            HERAM8 <= '0';
            HERAM16 <= '0';
            if(SELRAM = '1' and DIR = 63 and CONMUTA = '1') then
              DATMRAM <= "0000000000000000";
              DIR <= DIR + 1;
              HERAM16 <= '1';
              CONMUTA := '0';
              LISTO <= '0';
              ESTADO := 3;
            else
              LISTO <= '1';
            end if;
          else
            if SELRAM = '0' then
              HERAM8 <= '1';
            else
              HERAM16 <= '1';
            end if;
          end if;
when 3 => DIR <= DIR + 1;
          if DIR > 127 then
            ESTADO := 7;
          end if;
```

Figura 5.8: Tercera parte de la entrada de datos.

Cuando la unidad analizadora se encuentra en el estado 2, habilita la escritura de la memoria seleccionada por la señal *SELRAM*, ya sea para la RAM de 8 bits (*HERAM8*) o para la de 16 (*HERAM16*), y de esta forma almacenar el dato leído. Después de esto, espera en un ciclo hasta que la señal *LEE* sea 0. Cuando esto ocurre, la unidad analizadora coloca en 0 la señal *ACK* y deshabilita la escritura de las memorias. Después puede

ocurrir una de dos situaciones. Si aún no se han recibido los 1728 bytes desde la unidad distribuidora, la unidad analizadora coloca en 1 la señal *LISTO* y pasa al estado 4 para preparar la recepción de otro dato. Pero si ya se recibieron todos los datos entonces la señal *DATMRAM* se fija con un valor de 0000000000000000, se habilita la escritura de la memoria de 16 bits y se pasa al estado 3, en donde solo se incrementa el valor de la direccion (señal *DIR*) hasta rellenar con ceros las localidades de la memoria de 16 bits, desde la 64 hasta la 127. Esto es con el fin de poner en cero estas localidades de memoria, y así dejarlas listas para almacenar los histogramas calculados del segmento de imagen recibido.

Para saber si ya se recibieron los 1728 bytes, la unidad analizadora revisa los valores de las señales dentro de la sentecia *IF*. Lo que hace es determinar si el último dato recibido fué almacenado en la parte alta de la memoria de 16 bits, cuando la dirección tenía un valor de 63.

En la figura 5.9 se muestra la etapa en donde la unidad analizadora espera recibir otro dato. Esto puede notarse porque la unidad espera en un ciclo hasta que la señal *LEE* sea 1. Cuando esto ocurre, en la variable *DIRTEMP* guarda de manera temporal la dirección donde debe ser almacenado el dato recibido, que es el valor de la señal *DIR* más uno. Si el dato va dirigido a la RAM de 8 bits ($SELRAM = 0$), la unidad pasa al estado 5 para preparar el almacenamiento del dato. Pero si no es así, entonces debe verificar el valor de la variable *CONMUTA*, para determinar si el dato recibido va a la parte alta o baja de la localidad de memoria de 16 bits. Si esta variable tiene un valor de 1, entonces el dato recibido deberá de ser almacenado en la parte baja de la localidad de memoria apuntada (el valor 1 indica que ya se escribió un dato en la parte alta, por lo que el nuevo dato deberá ir en la parte baja). Pero si la variable tiene un valor de 0, entonces deberá de ir en la parte alta de la localidad de memoria. Es en este momento donde es necesario realizar un ajuste al valor de la dirección de memoria, ya que si el dato será almacenado en la parte alta de la localidad, hay que disminuir el valor de la dirección en uno. De esta

forma, el dato leído es almacenado en la misma localidad, pero en la parte alta. Cuando *SELRAM* tiene un valor de 1, la unidad analizadora debe pasar posteriormente al estado 6 para prepar el almacenamiento del dato recibido.

```
when 4 => if lee = '1' then
            DIRTEMP := DIR + 1;
            LISTO <= '0';
            if SELRAM = '0' then
              ESTADO := 5;
            else
              if CONMUTA = '0' then
                DIRTEMP := DIRTEMP - 1;
              end if;
              ESTADO := 6;
            end if;
          DIR <= DIRTEMP;
          end if;
```

Figura 5.9: Cuarta parte de la entrada de datos.

En este momento se puede apreciar la utilidad de las variables. Si hubieramos utilizado la señal *DIR* en lugar de la variable *DIRTEMP* no hubieramos tenido el resultado esperado. El uso de esta variable tuvo utilidad para direccionar la memoria de 16 bits. Para la memoria de 8 bits no tuvo utilidad, sin embargo su uso permitió tener una implementación más fácil.

La última parte de la recepción de datos aparece en la figura 5.10. En el estado 5, la unidad analizadora debe verificar si ya se recibieron todos los valores del segmento de imagen. Esto se logra al comparar el valor de la dirección actual (señal *DIR*) con 1599. Si es mayor, entonces los siguientes datos corresponden a los valores de los histogramas. Es por ello que cambia el valor de la variable *SELRAM* a 1, con el propósito de que las siguientes escrituras sean realizadas en la memoria de 16 bits. El valor de la variable *CONMUTA* es colocada a 1 en el estado 3. Esto es solo para inicializarla, ya que en el estado 6 es invertida y así poder iniciar el ciclo de escritura en la parte baja de la RAM de 16.

```
when 5 => HERAM8 <= '0';
        if DIR > 1599 then
          DATMRAM <= "0000000000000000";
          DIR <= "00000000000";
          HERAM8 <= '0';
          CONMUTA := '1';
          SELRAM := '1';
          ESTADO := 6;
        else
          ESTADO := 1;
        end if;
when 6 => HERAM16 <= '0';
        ESTADO := 1;
        CONMUTA := not CONMUTA;
```

Figura 5.10: Quinta parte de la entrada de datos.

5.2.2. Unidades de memoria

Como fué mencionado en la sección anterior, cada unidad analizadora contiene dos memorias RAM, una de 8 bits para almacenar los valores de los píxeles, y otra de 16 bits para almacenar los datos de los histogramas. Para definir estas memorias en VHDL, se tomó el modelo como es sugerido en el manual de referencia de Synplify [14]

En la figura 5.11 aparece la definición en VHDL de la RAM de 8 bits. Esta consiste de un arreglo de 1600 elementos de tipo std_logic_vector, de 8 elementos. Todos los puertos y señales de los componentes definidos para el proyecto son de tipo std_logic o std_logic_vector, ya que se utilizaron funciones que solo están definidas para este tipo de datos, como *conv_integer()* que aparece en la definición de la RAM. Es por ello que en todas las definiciones de los componentes se importa la biblioteca *IEEE*.

Para la RAM de 16 bits la definición es parecida, la diferencia es que en este caso solo hay 128 elementos de tipo std_logic_vector de 16 elementos.

5.2.3. Selección de las ventanas

Una vez que todos los datos se encuentran en las memorias, ahora la unidad analizadora está lista para empezar con los cálculos de los histogramas. Para ello, la unidad de control va seleccionando ventanas de 3x3 píxeles, con la finalidad de obtener los valores de brillo

```
--Memora RAM de 8 bits para los datos
--de los pixels de la imagen
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity RAM8 is
  port (
    DATO : in std_logic_vector(7 downto 0);
    LINDIR : in std_logic_vector(10 downto 0);
    ESCR : in std_logic;
    CLK : in std_logic;
    DATSAL : out std_logic_vector(7 downto 0));
end RAM8;

architecture RTL of RAM8 is
  type mem_type is array (1600 downto 0) of
                    std_logic_vector(7 downto 0);
  signal mem: mem_type;
begin
  process (CLK)
  begin
    if CLK'event and CLK='1' then
      if (ESCR = '1') then
        mem(conv_integer(LINDIR)) <= DATO;
      end if;
      end if;
  end process;
  DATSAL <= mem(conv_integer(LINDIR));
end RTL;
```

Figura 5.11: Definición de la memoria RAM 8.

y contraste correspondientes al pixel central, como se muestra en la figura 5.12. Aquí hay que aclarar que los valores de brillo y contraste solo se calculan para los píxeles del área sombreada, ya que no es posible calcularlos para los píxeles de la periferia al no estar rodeados por 8 píxeles vecinos.

Como se mencionó en la sección 5.2.1, los valores de los píxeles del segmento de imagen son recibidos por filas, empezando por la fila superior. Esto indica que los datos en las localidades 0 a 39 de la RAM de 8 bits corresponden a la primera fila, de las localidades 40 a 79 a la segunda fila, de la 80 a la 119 a la tercera fila, y así sucesivamente. Esto quiere decir que al momento de seleccionar una ventana, hay que tomar en cuenta que

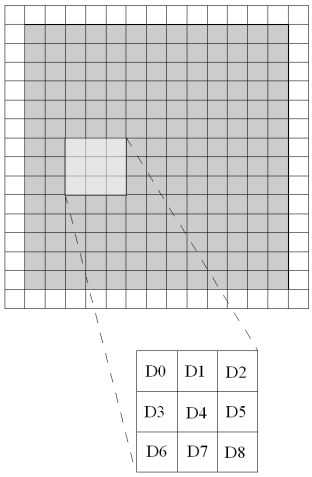


Figura 5.12: Selección de una ventana de 3x3 píxeles.

sus elementos no estan en celdas contiguas de la memoria.

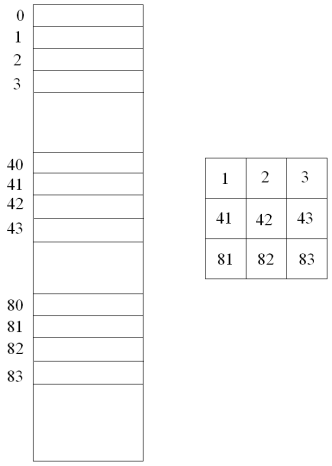


Figura 5.13: Selección de una ventana de 3x3 píxeles de la RAM de 16 bits.

Para generar las direcciones de la RAM y leer los datos de la ventana, tomamos como base la dirección del pixel central, a partir de esta es posible generar las direcciones para leer los valores de los píxeles vecinos. Por ejemplo, para el pixel en la posición 42, debemos recuperar los píxeles localizados en las posiciones 1,2,3,41,42,43,81,82 y 83. Este proceso está representado en la figura 5.13. La dirección del pixel central es 42. La dirección de dos píxeles vecinos se obtiene sumando y restando 1 a este valor (41 y 43). Para obtener la dirección de los píxeles superiores, restamos cuarenta a la dirección del pixel central, lo que nos da 42-40=2, y a este valor le podemos sumar y restar 1, para así obtener las

direcciones de los píxeles superiores (1,2,3). Para el caso de los píxeles inferiores, usamos un procedimiento parecido, solo que sumamos 40 en lugar de restar (42+40=82), al sumar y restar 1 obtenemos las otras direcciones (81,82,83).

```
when 7 => HERAM8 <= '0';
        HERAM16 <= '0';
        ACK <= '0'; -- fin del ultimo ack
        POSX := -1;
        POSY := -1;
        POSCX := 1;
        POSCY := 1;
        ESTADO := 8;
when 8 => DIRREG := "0000";
        ESTADO := 9;
        DIR <= conv_std_logic_vector(40 * (POSCY + POSY)
                + POSCX + POSX,11);
        SELREG <= DIRREG;
when 9 => DIRREG := DIRREG + 1;
        POSX := POSX + 1;
        if POSX > 1 then
          POSX := -1;
          POSY := POSY + 1;
          if POSY > 1 then
            POSY := -1;
          end if;
        end if;
        if DIRREG > 8 then
          ESTADO := 10;
        else
          DIR <= conv_std_logic_vector(40 * (POSCY + POSY)
                  + POSCX + POSX,11);
        end if;
        SELREG <= DIRREG;
when 10 => CONVIERTE <= '1';
         ESTADO := 11;
when 11 => CONVIERTE <= '0';
         if TERMINADO = '1' then
           VALCONT := conv_integer(CONTRASTE);
           ESTADO := 12;
         end if;
```

Figura 5.14: Selección de la ventana.

En la figura 5.14 se puede apreciar el segmento del código para la unidad de control, encargada de seleccionar esta ventana. La unidad de control llega al estado 7 después de que fueron cargados todos los datos en las memorias. En este estado se inicializan las variables necesarias para generar las direcciones. Cuando la unidad de control pasa al estado 8 utiliza la señal *SELREG* para seleccionar uno de nueve registros, el cual almacenará el valor del pixel direccionado (Estos registros son explicados en la siguiente

subsección). Las variables *POSCY* y *POSCX* determinan la posición vertical y horizontal, respectivamente, del pixel central de la ventana seleccionada dentro del segmento de imagen. Estos valores pueden variar de 1 a 38, ya que como se recordará no son usados los píxeles en la periferia. Las variables *POSY* y *POSX* indican si se suma 1, se resta 1 o permanece igual los valores de *POSCY* y *POSCX*. La suma *POSCY+POSY* nos indica en que fila esta el pixel direccionado, y este valor es multiplicado por 40 porque este es el número de elementos por fila. Para obtener la dirección del pixel, a este valor se le suma *POSCX+POSX*, que nos indica la columna donde se encuentra el pixel.

La unidad de control permanece en el estado 9, hasta que son cargados los nueve elementos de la ventana. Esto se logra al comparar el valor de la variable *DIRREG* con 8. Si es mayor, entonces ya fueron leidos los nueve datos de la ventana. Una vez hecho esto, en el estado 10 la unidad de control activa la señal *CONVIERTE*, que es usada para indicar a la unidad de cálculo que los datos de la ventana están listos, por lo que puede iniciar el cálculo del brillo y contraste del pixel central. Por último, la unidad de control permanece en el estado 11 hasta que la unidad de cálculo active la señal *TERMINADO*. En este momento la unidad de control lee los valores obtenidos.

5.2.4. Cálculo de los valores para brillo y contraste

Los valores de la ventana seleccionada son recibidos uno por uno por un grupo de 9 registros. Cada pixel es almacenado en un registro distinto. Su función es la de retener el valor de los píxeles, con el fin de que la unidad de cálculo determine el valor de brillo y contraste para el pixel central. Como aparece en la figura 5.15, cada registro retiene un valor de la ventana seleccionada. La señal *SELREG* es un valor de 4 bits, manejada por la unidad de control, y tiene la finalidad de determinar cual registro debe ser habilitado para almacenar el dato enviado. Esto se logra con la ayuda del decodificador, el cual genera la señal de habilitación correspondiente al valor de la señal *SELREG*.

Teniendo los valores de la ventana seleccionada en los registros, el cálculo del valor de

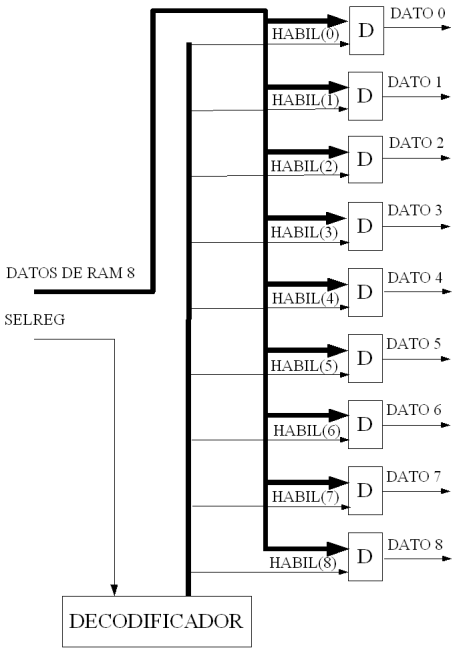
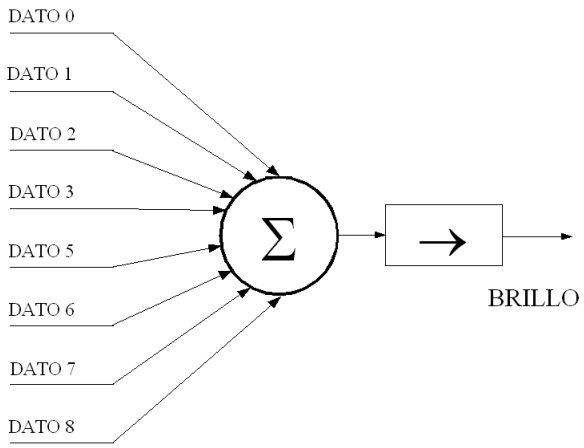


Figura 5.15: Arreglo para retener los valores de la ventana seleccionada.

brillo es sencillo. Basta enviar a las entradas de un sumador los valores de los píxeles vecinos. La salida de este sumador pasa por un registro de corrimiento, con la finalidad de dividir la suma entre 8 y así obtener el promedio. Esto queda representado en la figura 5.16, en donde los valores de entrada están referidos a los mostrados en la figura 5.12. Por ser un circuito combinacional, el valor para el brillo permanecerá mientras los valores de la ventana no cambien. En la misma figura aparece el código de VHDL usado para obtener el brillo. La señal *SALIDAB*, es la que conduce el valor para el brillo. Como se puede ver, se utiliza el operador de división para obtener el promedio, sin embargo no es instanciado ningún divisor. La herramienta de síntesis Synplify se encarga de analizar el código e instanciar los componentes más apropiados, en este caso un registro de corrimiento.

El cálculo del valor de contraste es más complicado, ya que es necesario seleccionar los píxeles mayores o menores al pixel central, realizar una suma de sus valores para posteriormente obtener su promedio. Después es necesario un proceso parecido para obtener el promedio de los píxeles menores. Finalmente, con estos promedios se puede obtener el valor de contraste. En el caso del brillo el cálculo del promedio fué sencillo, porque hay que




```
SUMBRI <= conv_integer(CUADRO1) + conv_integer(CUADRO2) +
          conv_integer(CUADRO3) + conv_integer(CUADRO4) +
          conv_integer(CUADRO6) + conv_integer(CUADRO7) +
          conv_integer(CUADRO8) + conv_integer(CUADRO9);
SALIDAB <= conv_std_logic_vector(SUMBRI/8,8);
```

Figura 5.16: Circuito para obtener el valor de brillo.

realizar siempre una división entre 8, que puede ser realizada por medio de un corrimiento. Para el contraste no es lo mismo, porque es necesario realizar una división entre un número que puede variar de 0 a 8. Esto obligó a definir un divisor para calcular los promedios necesarios. De hecho, es principalmente este divisor el que determina el tiempo que tarda la unidad en generar los valores, ya que fué diseñado usando un circuito secuencial, a diferencia de los demás componentes de la unidad de cálculo que son combinatorios.

En la figura 5.17 aparece la primera etapa para el cálculo del contraste. Se trata de 8 comparadores con dos entradas, en donde una de las entradas es el valor del pixel central (marcada como *DATO 4* en la figura), y el valor de cada pixel vecino (*DATO 0* a *DATO 8*) es la entrada de su respectivo comparador. Las salidas de los comparadores son las señales *IND(X)*, que tienen un valor de 1 si el valor del pixel vecino es mayor o igual al pixel central, y un valor de 0 en caso contrario. Por ejemplo, si el valor de *DATO 0* es mayor o igual que *DATO 4*, entonces la señal *IND(0)* tiene un valor de 1. Las señales *IND(X)* controlan la salida de un primer grupo de 8 multiplexores. Aquellos multiplexores cuya señal *IND(X)* sea 1, tienen a la salida el valor del pixel vecino respectivo. Volviendo al caso en el que *DATO 0* es mayor que *DATO 4,* el multiplexor al que está conectado

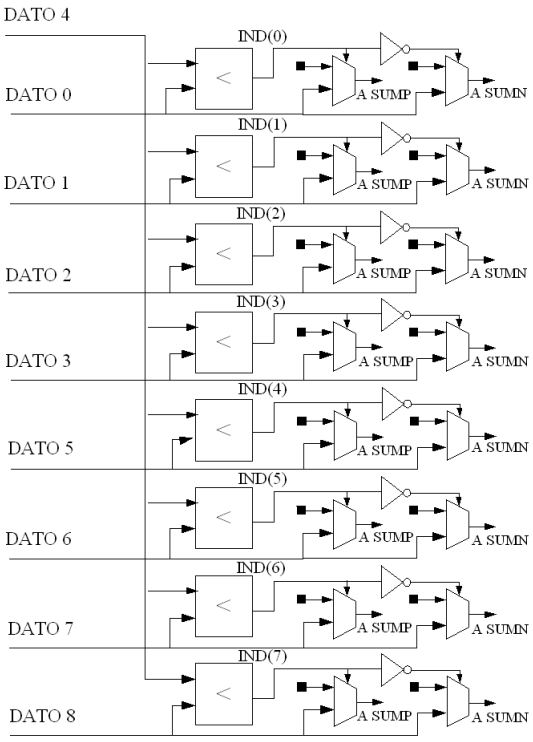


Figura 5.17: Primera etapa para el cálculo del valor de contraste.

IND(0) tiene a la salida el valor de *DATO 4*. Los multiplexores cuya señal *IND(X)* sea cero, presentan a la salida un valor de 0. Como resultado, en las salidas de este primer grupo de 8 multiplexores, tenemos los valores de los píxeles vecinos que son mayores o iguales al pixel central, más un valor de cero en aquellos casos donde sean menores. Las salidas de este primer grupo de multiplexores son las entradas de un sumador (marcado como SUMP), que nos dará la suma de los píxeles con valores mayores o iguales al pixel central (mostrado mas adelante).

De manera semejante es posible seleccionar los píxeles con valores menores al pixel central. Para esto se hace pasar las señales *INT(X)* a través de unos inversores, y posteriormente usar estas señales invertidas para controlar la salida de un segundo grupo de multiplexores. Para el caso descrito anteriormente, tenemos que *IND(0)* es 1 porque *DATO 0* es mayor que *DATO 4*, al pasar *IND(0)* por el inversor se convierte en 0, y es por esto que en la salida del multiplexor del segundo grupo aparece un 0 a la salida. En este segundo grupo de multiplexores, en las salidas estarán los valores de los píxeles cuyo

valor sea menor al pixel central, y 0 en los demás casos. Como en el caso anterior, estas salidas son las entradas de un segundo sumador, llamado SUMN, que nos da la suma de los valores de los píxeles menores al pixel central.

Las señales *IND(X)* tienen otra utilidad. Si tomamos el número de 1's que hay en estas señales, tenemos el número de píxeles mayores o iguales al pixel central. Este valor es usado junto con la salida del sumador SUMP para obtener el valor promedio, con la ayuda de uno de los divisores. Para el caso de los píxeles con valor menor al pixel central, a un valor constante de 8 le restamos el número de 1's que hay en las señales *IND(X)*, y este valor junto con la salida del sumador SUMN son las entradas del otro divisor para obtener el otro promedio necesario. Esto está representado en la figura 5.18

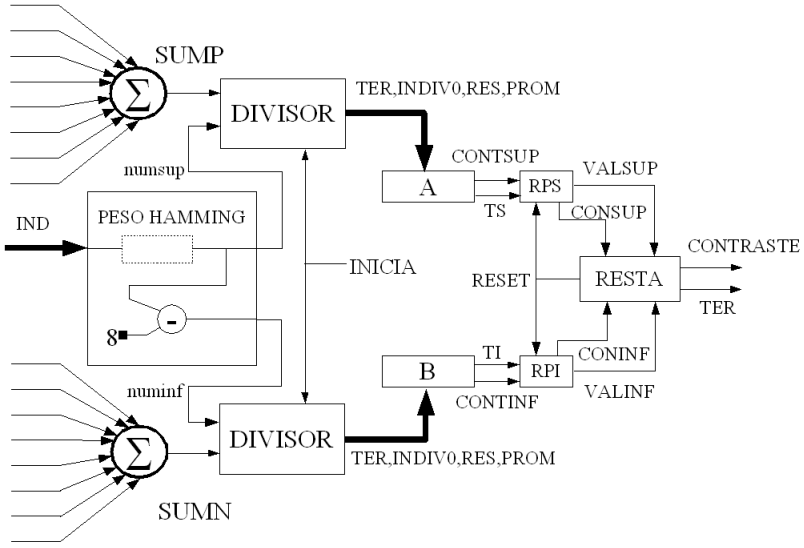


Figura 5.18: Segunda etapa para el cálculo del valor de contraste.

El módulo marcado como Peso Hamming, es el encargado de determinar el número de 1's en las señales *IND(X)*. La señal *INICIA*, manejada por la unidad de control, es utilizada para indicar a los divisores que los datos que tiene a la entrada son válidos y puede empezar a realizar la división. Esta señal es activada en el siguiente pulso de reloj después de cargar los valores de la ventana, o sea, después de que se carga el valor *DATA 8* en el último de los registros (figura 5.15). Hasta este momento, el valor de las sumas y los números de píxeles fueron obtenidos de circuitos combinatorios. Es por ello que para

tener listos los datos de los divisores, la frecuencia de reloj debe ser seleccionada de tal forma que un pulso de reloj sea suficiente.

A diferencia del resto de los circuitos de la unidad de cálculo, para el desarrollo de los divisores fué utilizado un diseño secuencial. La descripción del funcionamiento del divisor, así como el código VHDL que lo implementa, se encuentra en el apéndice A. Para obtener la mayor rapidez en el cálculo del valor de contraste, se utilizaron dos divisores independientes, uno para cada promedio a obtener. La señal *TER* de los divisores es para indicar que ya terminó el calculo de la división, y el promedio encuentrado está en la señal *PROM*, y el residuo de la división está en *RES*. La señal *INDIV0* de los divisores indican si los operadores introducidos dieron origen a una división por cero.

Como cada divisor trabaja de manera independiente, y puede darse el caso de que uno termine antes que el otro. La función de los módulos A y B es realizar la corrección necesaria cuando haya una división entre cero (esto es, dar como promedio el valor del pixel central, si todos los píxeles son mayores o iguales al pixel central, o si todos son menores), redondear el valor de la división, y finalmente indicar a los componentes RPS y RPI que retengan el valor obtenido. Para el caso de los píxeles mayores o iguales al pixel central, el módulo A realiza las correciones necesarias cuando el divisor termina de efectuar su cálculo. La señal *TS*, que sale del módulo A, indica al componente RPS que retenga el valor del promedio obtenido (la señal *CONTSUP*), ya que una vez que el divisor obtiene su resultado, al siguiente pulso de reloj ya no es válido. La función del módulo RPS es retener este valor hasta que ambos divisores terminen. Por su parte, el componente RPS coloca la señal *VALSUP* en 1 para indicar que su salida *CONSUP* contiene el valor del promedio buscado, y permanecerá así hasta que la señal *RESET* sea colocada en 1. Para el caso de los píxeles menores al pixel central, el módulo B y el componente RPI realizan funciones parecidas, y cuando la señal *VALINF* es colocada en 1 indica que *CONINF* contiene el segundo promedio.

Una vez que ambas señales *VALSUP* y *VALINF* son 1, indicando que el resultado

de los promedios es válido, el módulo RESTA lee los valores de *CONSUP* y *CONINF*, para realizar la resta del promedio de píxeles mayores o iguales al pixel central, menos el promedio de píxeles menores. El resultado de esta resta es el valor del contraste buscado. En el siguiente pulso de reloj, el mismo módulo RESTA activa la señal *RESET*, dejando a los componentes RPS y RPI listos para recibir otros valores. El valor del contraste permanece solo por un ciclo de reloj, y la señal *TER* es activada para indicar que este valor es válido. Es en este momento cuando la unidad de control lee los valores de brillo y contraste, para elaborar el histograma.

5.2.5. Generación del histograma

Para cada ventana seleccionada, la unidad de cálculo genera los valores de brillo y contraste para el pixel central. Estos valores son recibidos por la unidad de control, con el fin de generar el histograma. Para ello hay que tomar en cuenta que los valores generados, tanto de brillo como de contraste, pueden variar de 0 a 255. Los histogramas deben de tener 32 intervalos, es por ello que los valores generados deben ser divididos entre 8. Estos últimos valores sirven como índice para seleccionar la celda de memoria adecuada.

Refiriéndonos a la figura 5.19, la memoria RAM de 16 bits es dividida en dos partes. La parte mas baja fué usada para almacenar los datos correspondientes a los histogramas de la imagen base. Los histogramas de la imagen de prueba se almacenan en la parte mas alta. A partir de la localidad 64 serán almacenados los datos del histograma de brillo, y a partir de la 96 los de contraste. Estas dos localidades de memoria se usan como base, a las que se les debe de sumar el índice respectivo, obtenido a partir de los valores de brillo y contraste generados por la unidad de cálculo. El resultado de estas sumas es un par de valores, uno por cada histograma, que apuntan a la localidad de memoria correspondiente al intervalo respectivo. Para cada localidad, se incrementa en uno el valor almacenado.

Como ejemplo, supongamos que para la ventana seleccionada el valor de brillo fue 193 y el de contraste 58. Estos valores son divididos entre 8, eliminando el residuo. Estos

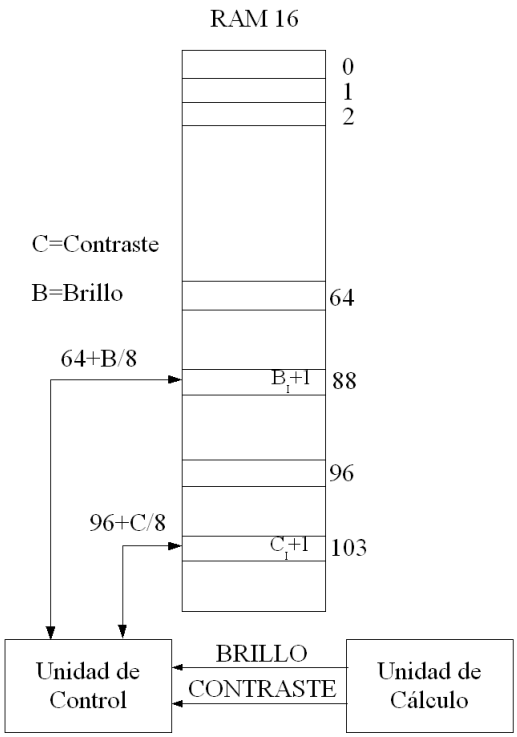


Figura 5.19: Selección de las localidades de memoria.

nos da 24 para el brillo y 7 para el contraste. Estos valores son los índices para calcular la dirección de la celda de memoria a incrementar. Para el brillo esta dirección es 64 + 24 = 88, y para el contraste es 96 + 7 = 103. Por último, los valores almacenados en las direcciones 88 y 103 son incrementados en 1. Este proceso continua hasta que son procesados todos los píxeles del segmento de imagen analizado.

En la figura 5.20 aparece el segmento de código VHDL que realiza esta labor. En las variables *BINBRI* y *BINCONT* son almacenados los valores de los intervalos correspondientes, y son utilizados como índices para obtener el valor de la dirección de memoria adecuada. Primero se utiliza *BINBRI* para localizar la localidad de memoria. Esta dirección es calculada en el estado 12. En el estado 13 el contenido de la localidad de memoria es incrementado. La señal *DATRAMC* es el salida de datos de la RAM de 16 a la unidad de control. La señal *DATMRAM* es la entrada de datos a la misma memoria. En el estado 14 se usa la variable *BINCONT* como índice para direccionar la otra localidad de memoria, correspondiente al histograma de contraste. En el estado 15 es incrementado el valor

de esta localidad de memoria.

```
when 12 => ESTADO := 13;
        BINBRI := conv_integer(BRILLO)/8;
        BINCONT := VALCONT/8;
        DIR <= conv_std_logic_vector(64 + BINBRI,11);
when 13 => BYTERAM := conv_integer(DATRAMC);
        BYTERAM := BYTERAM + 1;
        DATMRAM <= conv_std_logic_vector(BYTERAM,16);
        HERAM16 <= '1';
        ESTADO := 14;
when 14 =>DIR <= conv_std_logic_vector(96 + BINCONT,11);
        HERAM16 <= '0';
        ESTADO := 15;
when 15 =>BYTERAM := conv_integer(DATRAMC);
        BYTERAM := BYTERAM + 1;
        DATMRAM <= conv_std_logic_vector(BYTERAM,16);
        HERAM16 <= '1';
        ESTADO := 16;
when 16 => ESTADO := 8;
        HERAM16 <= '0';
        POSCX := POSCX + 1;
        if POSCX > 38 then
          POSCX := 1;
          POSCY := POSCY + 1;
          if POSCY > 38 then
            ESTADO := 17;
            POSCY := 1;
          end if;
        end if;
```

Figura 5.20: Selección de las localidades de memoria.

En el estado 16 se modifica el valor de las variables *POSCX* y *POSCY* para seleccionar otra ventana, verificando que no sobrepasen el valor de 38. Hay que recordar que estas variables indican la localización del pixel central de la ventana analizada, dentro del segmento de imagen. Si *POSCX* sobrepasa el valor de 38 quiere decir que ya se terminó de analizar una línea del segmento de imagen, por lo que se incrementa el valor de *POSCY*. Cuando esta última variable sobrepase el mismo valor, quiere decir que ya fueron analizados todos los píxeles del segmento, por lo que el histograma de la imagen de prueba está completo. En este momento se puede proceder a realizar la comparación de los histogramas.

5.2.6. Comparación de histogramas

Después de procesar todos los píxeles del segmento de imagen, tenemos en la memoria RAM de 16 bits los valores de los histogramas buscados. Como se muestra en la figura 5.21, de las localidades 0 a 31 está el histograma de brillo de la imagen base, de la 32 a la 63 está el histograma de contraste igualmente de la imagen base, de la 64 a la 95 el histograma de brillo de la imagen de prueba, y finalmente de la 96 a la 127 el de contraste. Lo que resta es realizar la comparación de los histogramas, tal como fué descrito en el capítulo 3. Para ello la unidad de control primero realiza la comparación de los histogramas de brillo. Lee los valores almacenados en las localidades 0 y 64, toma el menor de estos valores y lo guarda en una variable llamada *SUMMINB*. Después lee los valores en las localidades 1 y 65, el menor lo suma a la variable *SUMMINB*. De esta forma continua hasta que llega a las localidades 31 y 95. Para el caso del contraste realiza un proceso parecido, solo que inicia a partir de las localidades 32 y 96, y los valores se van acumulando en la variable *SUMMINC*. Como ejemplo, haciendo referencia a la misma figura 5.21, y para el caso del histograma de brillo, en la localidad 0 está almacenado el valor 5, y en la localidad 64 está el valor 2, por lo que al contenido de la variable *SUMMINB* se le suma el valor de 2. En la figura también se muestra la comparación para el caso del histograma de contraste, en este caso son comparados los contenidos de las localidades 32 y 96. El valor mínimo, cero en este caso, es sumado al valor de la variable *SUMMINC*.

Una vez obtenidos las sumas de los valores mínimos, lo que resta es utilizar la fórmula para obtener el nivel de semejanza entre las imágenes analizadas. Un vez obtenida la semejanza, la unidad analizadora indica esto manteniendo en valor 1 por un ciclo de reloj la señal *FIN*, y si las imagenes son semejantes, la señal *SEMEJA* tendrá un valor de 1, y en caso contrario este valor será de cero.

En la figura 5.22 está el código de VHDL que realiza la comparación de histogramas. Primero en el estado 17 se inicializan las variables necesarias. En la variable *SUMMINB* se acumulan los valores mínimos del histogramas de brillo, y en la variable *SUMMINC* los

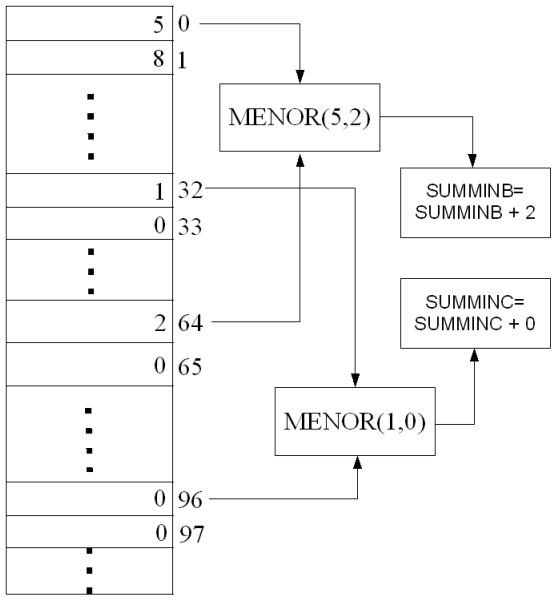


Figura 5.21: Suma de valores mínimos de los histogramas.

de contraste. La variable *TEMP* es usada como un índice. La constante *DIRREF* apunta a la dirección base de los histogramas de la imagen base, y *DIRCAL* a la dirección base de los histogramas calculados. La variable *TEMP* tiene inicialmente un valor de 0, por lo que en el estado 18 direccionamos la primera localidad del histograma de brillo, y el valor leído se almacena en la variable *VALORR* en el estado 19. En este mismo estado se cambia la dirección, para apuntar a la localidad de memoria donde está el primer valor calculado del histograma de brillo, y posteriormente asignarlo a la variable *VALORC* en el estado 20. Aquí es necesario revisar el valor de la variable *TEMP* para determinar el histograma con el que se está trabajando. Si el valor de *TEMP* está entre 0 y 31, entonces se están leyendo los valores de los histogramas de brillo, y los valores mínimos son acumulados en la variable *SUMMINB*. Pero si el valor de *TEMP* está entre 32 y 63, entonces se están leyendo los valores del histograma de contraste, y los valores mínimos ahora deben ser acumulados en la variable *SUMMINC*.

Cuando el valor de *TEMP* es 64 ya fueron comparados los histogramas de brillo y contraste, y ya se tienen las sumas de los valores mínimos de cada uno de ellos. Por lo que ahora es necesario utilizar la fórmula para determinar el nivel de semejanza entre los

```
when 17 => SUMMINB := 0; -- Suma de valores minimos de brillo
        SUMMINC := 0; -- Suma de valores minimos de contraste
        TEMP := 0;   -- Indice para direcciones
        ESTADO := 18;
when 18 => DIR <= conv_std_logic_vector(DIRREF + TEMP,11);
        ESTADO := 19;
when 19 => DIR <= conv_std_logic_vector(DIRCAL + TEMP,11);
        VALORR := conv_integer(DATRAMC);
        ESTADO := 20;
when 20 => TEMP := TEMP + 1;
        VALORC := conv_integer(DATRAMC);
        ESTADO := 18;
        case TEMP is
          when 0 to 31 =>
            if VALORR > VALORC then
              SUMMINB := SUMMINB + VALORC;
            else
              SUMMINB := SUMMINB + VALORR;
            end if;
          when 32 to 63 =>
            if VALORR > VALORC then
              SUMMINC := SUMMINC + VALORC;
            else
              SUMMINC := SUMMINC + VALORR;
            end if;
          when 64 =>
            ESTADO := 0;
            if ((SUMMINB*4)+(SUMMINC*6)) > 10500 then
              SEMEJA <= '1';
            else
              SEMEJA <= '0';
            end if;
            FIN <= '1';
          when others => null;
        end case;
```

Figura 5.22: Código usado para la comparación de histogramas.

histogramas. Si resulta que ambos histogramas son parecidos, a la señal *SEMEJA* se le asigna un valor de 1, pero si son distintos se le asigna un valor de 0. Por último, a la señal *FIN* se le asigna un valor de 1, y permanece así por un ciclo de reloj, momento que se debe aprovechar para leer el valor de *SEMEJA*.

Finalmente, la unidad de control pasa al estado 0, en donde espera en un ciclo hasta que empiece la recepción de otro segmento de imagen.

Capítulo 6

Resultados y conclusiones

En esta última sección se muestran los tiempos de cálculo obtenidos con la unidad analizadora, con la que se puede estimar el tiempo necesario para procesar una imagen, así como el espacio en FPGA necesario, información útil para determinar si la arquitectura propuesta es viable. Además se explican los trabajos pendientes a realizar, para poder obtener la parte de procesamiento de imagen del sistema de inspección.

6.1. Resultados

Para probar la efectividad del algoritmo de comparación de imágenes, explicado en el capítulo 3, se tomaron imágenes de una tarjeta con pasta depósitada. Las imágenes usadas son de 640 x 480 píxeles, obtenidas por medio de un escáner. En la figura 6.1 se muestra la ventana del programa desarrollado, con la imagen base y la de prueba. Como se puede apreciar, al algoritmo funciona bien en áreas con iluminación homogénea, existiendo un problema en áreas con luz especular. Esto último provocaría que dos imágenes semejantes sean detectadas erróneamente distintas. Este problema puede resolverse usando un sistema que ilumine las tarjetas de forma homogénea. El desarrollo de este sistema de iluminación queda para trabajo futuro.

El trabajo de esta tesis fué la del diseño del circuito básico para procesamiento de

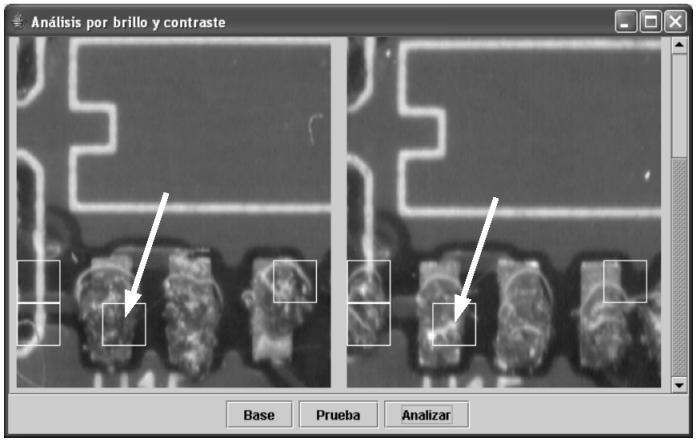


Figura 6.1: Imágenes mostrando efectos de la luz especular(partes brillosas marcadas).

imágenes, y que está representada por la unidad básica analizadora explicada en el cápitulo 5. Con el diseño de este circuito se obtuvo el tiempo necesario para obtener los histogramas de un segmento de imagen, y su posterior comparación con los histogramas de la imagen base. Este resultado se usó para estimar el tiempo de ejecución necesario para analizar una imagen completa. Además, fué posible estimar el espacio en FPGA necesario para implementar el procesamiento de imagen.

Para conocer estos datos, se utilizó la herramienta de síntesis Synplify. Después de realizar la síntesis, se determinó que el circuito diseñado puede trabajar a una frecuencia de 25 Mhz con el FPGA Spartan III, utilizado para realizar los experimentos. Sin embargo, al utilizar la herramienta de sínteis con FPGA's de la familia Virtex II, se determinó que el circuito puede trabajar a 50 Mhz. Para conocer el tiempo de procesamiento, el diseño del circuito fué simulado utilizando ModelSim, encontrado un tiempo de procesamiento de 1.8 milisegundos, trabajando el circuito a la frecuencia de 25 Mhz, y de 0.9 milisegundos trabajando a 50 Mhz.

Usando la misma herramienta de síntesis se puede estimar el espacio de FPGA que se necesita. En este punto hay que tomar en cuenta que se desea tener un sistema de inspección independiente, de tal forma que en un solo FPGA se puedan realizar todas las operaciones necesarias para su funcionamiento. También se desea tener varias unidades

analizadoras trabajando en paralelo, cada una sobre diferentes segmentos de la imágen. Es por ello que el FPGA usado debe tener espacio suficiente para tener varias unidades analizadoras, más el necesario para las demás funciones.

Para determinar el número de unidades necesarias, hay que tener presente que el sistema debe procesar las imágenes en tiempo real. Esto quiere decir que debe ser capaz de procesar 30 imágenes por segundo, o aproximadamente 33 milisegundos por imagen. Cada imagen de 640x480 píxeles tiene 12 filas de segmentos de 40x40 píxeles, lo que nos quiere decir que cada fila debe ser procesada en $33 \div 12 = 2.75$ ms. Es por ello que, para el sistema de inspección, se propone el uso de 2 grupos de 16 unidades analizadoras básicas (ya que este es el número de segmentos por fila), cada uno actuando sobre una fila diferente de la imagen. Así, cuando el primer grupo termina de recibir los datos para ser procesados, se puede empezar el envío al segundo grupo. Puesto que las 16 unidades trabajan de forma paralela, cada fila puede ser procesada en 0.9 milisegundos trabajando a 50 Mhz. Esto quiere decir que antes de que el segundo grupo de unidades termine de procesar la fila asignada, el primer grupo ya terminó de procesar la fila enviada, por lo que está lista para recibir otra. Debido a lo anterior, el FPGA debe tener espacio para 32 unidades básicas.

Existen FPGA con los que podemos tener el espacio requerido, como los Virtex II y Virtex II pro. Estos circuitos cuentan con recursos adicionales a los bloques programables, semejantes a los explicados en el capítulo 4 para el caso del Spartan III. Sobre todo hay circuitos con bloques de memoria suficiente para programar las 32 unidades básicas necesarias. Tal es el caso del XC2V4000, en donde las 32 unidades ocuparían el 32 % de espacio, dejando el suficiente para manejar las demás funciones del sistema de inspección.

A continuación, se presentan algunos datos obtenidos como resultado de la síntesis de una unidad analizadora:

- Frecuencia estimada: 65 mhz
- Bloques de RAM usados: 2 de 120.

- Total de LUT´s usados: 972 de 46080.

Como se puede apreciar, el FPGA XC2V4000 puede trabajar a la frecuencia necesaria para procesar imágenes en tiempo real. Además, cuenta con los bloque de memoria necesarios, así como los circuitos lógicos para implementar las 32 unidades planeadas.

6.2. Conclusiones

Como conclusión de esta tesis podemos decir que fué elaborado un algoritmo para comparación de imágenes, en base a histogramas que representan las características de textura. El diseño de este algoritmo permitió una fácil implementación en FPGA.

Además se diseñó el circuito básico para implementar el algoritmo. Con este diseño fué posible estimar los tiempos de ejecución, resultando que es posible realizar el análisis de imágenes en tiempo real, tiempo requerido para la inspección de tarjetas.

Como resultado de la síntesis del circuito, existe la posibilidad de manejar todo el sistema de inspección con un solo FPGA, por lo que puede ser posible tener un sistema independiente.

6.3. Trabajo a futuro

Al momento de realizar esta tesis aún no se contaba con la etapa de captura de imagen, por lo que aún hace falta determinar el tiempo necesario para la distribución de datos, que comprende el envío de los segmentos de imagen e histogramas a las unidades analizadoras. Para cumplir con las restricciones de tiempo es necesario modificar el protocolo de entrada de datos, ya que este fué diseñado solo para pruebas.

Las pruebas del algoritmo indican que puede funcionar para las imágenes con las que debe trabajar. Hace falta diseñar el sistema de iluminación para eliminar los problemas de luz especular, y así poder analizar una mayor cantidad de imágenes.

Además de modificar el protocolo de entrada de datos, también se puede experimentar con otros tipos de algoritmos de división. Con el algoritmo empleado, el tiempo en obtener el resultado es proporcional al número de bits del dividendo. Como el tiempo de cálculo depende principalmente del utilizado por los divisores, el disminuir el número de ciclos para la división reduciría el tiempo de cálculo.

Además de la colocación de la soldadura en pasta, otro aspecto importante es la cantidad aplicada de la misma. En este trabajo se presenta un sistema para inspeccionar la correcta colocación de la soldadura, pero hace falta el análisis respectivo para determinar el volúmen. Esto último solo se puede lograr con un sistema de inspección en 3 dimensiones. Hace falta entonces la elaboración de un algoritmo apropiado para realizar este análisis.

Apéndice A

Componentes y código de la unidad analizadora

En esta parte se explican los componentes que forman la unidada analizadora, y que no fueron explicados en el capítulo 5.

A.1. Divisor

El algoritmo utilizado para crear el divisor es el de división con restauración. El algoritmo fué adaptado para dividir dos números enteros sin signo. A continuación se enumeran los pasos a seguir, considerando que el registro usado para el residuo contiene ceros.

1. Desplazar un bit a la izquierda el residuo
2. El bit más significativo del dividendo se copia al bit menos significativo del residuo.
3. Desplazar un bit a la izquierda el dividendo.
4. Al residuo restarle el valor del divisor.
5. Si el residuo es un número negativo (su bit más significativo es 1), entonces al bit menos significativo del dividendo se le asigna 0, y al residuo se le suma el valor del

divisor (se restaura). Pero si el residuo es positivo, entonces al bit menos significativo del dividendo se le asigna 1.

6. Regresar al punto 1 hasta completar n pasos, siendo n el número de bits del dividendo.

Como se puede notar, en el paso 5 es necesaria realizar una resta con signo. Esto es importante porque así lo requiere el algoritmo. Sin embargo, el valor del dividendo y el divisor deben ser enteros positivos, siendo el dividendo mayor o igual al divisor.

En el algoritmo se usa el mismo registro para almacenar el valor del dividendo y del resultado. Este último se va insertando desde la derecha en cada paso. Al final, después de los 11 pasos, el resultado está en los 8 bits menos significativos del dividendo, esto es tomando en consideración el tamaño del divisor y dividendo usados en el proyecto.

En nuestro caso, el dividendo contiene la suma de los píxeles vecinos, que como máximo puede ser $8 \times 255{=}2040$, por lo que el dividendo tiene 11 bits. El divisor puede contener valores de 0 a 8, dependiendo del número de píxeles vecinos mayores o iguales al pixel central. Esto implica usar 4 bits, pero como el residuo es usado en VHDL para realizar operaciones con signo, esto hace necesario definirlo de 5 bits. El algoritmo descrito es proporcional al número de bits del dividendo, y como este tiene 11, igualmente este es el número de pasos para obtener el valor del resultado.

En la figura A.1 aparece la primera parte del código usado para implementar el divisor en VHDL. El dividendo debe estar en la señal de entrada *OPMAY*, y el divisor en *OPMENOR*. Estos nombres hacen énfasis en que el dividendo debe ser mayor al divisor. La señal *INICIO*, manejada por la unidad de control, sirve para indicar que inicie el cálculo de la división. Esto se puede ver en el estado 0, en donde el divisor espera a que *INICIO* sea 1.

En la figura A.2 muestra la segunda parte del código. Una vez que la señal *INICIO* es 1, el divisor pasa al estado 1. En este estado checa el valor del divisor. Si es cero, entonces son borradas las salidas, se coloca en 1 la señal *DIVX0* con el fin de indicar una división

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

entity DIVISION is
  port (
    OPMAY     : in std_logic_vector(10 downto 0);
    OPMENOR   : in std_logic_vector(4 downto 0);
    INICIO    : in std_logic;
    FIN       : out std_logic;
    DIVX0     : out std_logic;
    CLK       : in std_logic;
    RESIDUO   : out std_logic_vector(4 downto 0);
    RESULTADO : out std_logic_vector(7 downto 0));

end DIVISION;

architecture DIV of DIVISION is

begin  -- DIV
DIVI: process
   variable DIVIDENDO : std_logic_vector(10 downto 0);
   variable DIVISOR : std_logic_vector(4 downto 0);
   variable R : std_logic_vector(4 downto 0);
   variable CONT : integer range 0 to 11 := 0;
   variable ESTADO : integer range 0 to 2 := 0;
   begin
     wait until CLK'event and CLK='1';
     case ESTADO is
       when 0 => FIN <= '0';
                 RESULTADO <= "00000000";
                 RESIDUO <= "00000";
                 DIVX0 <='0';
                 if INICIO = '1' then
                   ESTADO := 1;
                 end if;
```

Figura A.1: Código del divisor, primera parte.

entre cero, se coloca en 1 la señal *FIN* para señalar la finalización del cálculo, y finalmente el divisor regresa al estado 0. Pero si el divisor es diferente de 0, entonces en el estado 2 se realizan todos los pasos descritos en el algoritmo. Al final se muestran en la salida el valor del resultado de la división, así como el residuo. Por medio de la señal de salida *FIN* se indica que estos valores son válidos. Finalmente el divisor regresa al estado 0, listo para realizar otra operación.

```
        when 1 => DIVIDENDO := OPMAY;
                  DIVISOR := OPMENOR;
                  if conv_integer(DIVISOR) = 0 then
                    RESULTADO <= "00000000";
                    RESIDUO <= "00000";
                    DIVX0 <= '1';
                    FIN <= '1';
                    ESTADO := 0;
                  else
                    R := "00000";
                    CONT := 0;
                    ESTADO := 2;
                  end if;
      when 2 => R(4 downto 1) := R(3 downto 0);
                R(0) := DIVIDENDO(10);
                DIVIDENDO(10 downto 1) := DIVIDENDO(9 downto 0);
                R := R - DIVISOR;
                if R(4) = '1' then
                  R := R + DIVISOR;
                DIVIDENDO(0) := '0';
                else
                  DIVIDENDO(0) := '1';
                end if;
                CONT := CONT + 1;
                if CONT = 11 then
                  RESIDUO <= R;
                  RESULTADO <= DIVIDENDO(7 downto 0);
                  FIN <= '1';
                  ESTADO := 0;
                end if;
       when others => null;
     end case;
   end process;
end DIV;
```

Figura A.2: Código del divisor, segunda parte.

A.2. Peso Hamming

El peso Hamming de un número binario es el número de unos que contiene. Como se explicó en el capítulo 5, este módulo es usado para determinar el numero de píxeles mayores o iguales al pixel central. Restandole a 8 este valor tenemos el número de píxeles menores al pixel central. En la figura A.3 está el código mostrando el proceso usado para obtener este componente. Recibe 8 señales de entrada de un solo bit cada una (señales *BITS0* a *BITS7*). La señal *INI* es la misma que la empleada para los divisores. Sin embargo, al ser este módulo de tipo combinatorio y con la frecuencia de reloj apropiada, las salidas estarán listas antes de que los divisores empiecen sus cálculos. Como se puede apreciar,

```
process(BITS0,BITS1,BITS2,BITS3,BITS4,BITS5,BITS6,BITS7,INI)
  variable NUM1 : std_logic_vector(4 downto 0);
begin
  if INI = '1' then
    NUM1 := "00000";
    if BITS0 = '1' then
      NUM1 := NUM1 + 1;
    end if;
    if BITS1 = '1' then
      NUM1 := NUM1 + 1;
    end if;
    if BITS2 = '1' then
      NUM1 := NUM1 + 1;
    end if;
    if BITS3 = '1' then
      NUM1 := NUM1 + 1;
    end if;
    if BITS4 = '1' then
      NUM1 := NUM1 + 1;
    end if;
    if BITS5 = '1' then
      NUM1 := NUM1 + 1;
    end if;
    if BITS6 = '1' then
      NUM1 := NUM1 + 1;
    end if;
    if BITS7 = '1' then
      NUM1 := NUM1 + 1;
    end if;
  end if;
  NUMBIT1 <= NUM1;
  NUMBIT0 <= 8-NUM1;
end process;
end CUENTA;
```

Figura A.3: Cálculo del peso hamming.

se trata de una serie de sentencias *if* donde la variable *NUM1* se incrementa por cada 1 que aparece en las señales de entrada.

A.3. Unidad de cálculo de brillo y contraste

Esta unidad recibe una ventana de 3x3 píxeles, para calcular los valores de brillo y contraste del pixel central. El funcionamiento de esta unidad fué explicado en la unidad 5. Aqui se explica el código VHDL usado para implementarla. Esta unidad tiene 8 entradas, *CUADRO0* a *CUADRO8*, una para cada pixel de la ventana. La señal de entrada *INI* es la señal enviada por la unidad de control para iniciar con el cálculo. El valor del brillo y

contraste cálculados se encuentran en las señales *SALIDAB* y *SALIDAC*, respectivamente. Los valores en esta sálidas son válidas cuando la señal *TER* tiene un valor de 1. En la figura A.4 está la definición de este componente.

```
entity UCBC is
port
-- Las entradas CUADROX son los valores de los nueve pixels de la ventana. La
-- entrada CUADRO4 es el valor del pixel central.
  (CUADRO0 : in std_logic_vector(7 downto 0);
   CUADRO1 : in std_logic_vector(7 downto 0);
   CUADRO2 : in std_logic_vector(7 downto 0);
   CUADRO3 : in std_logic_vector(7 downto 0);
   CUADRO4 : in std_logic_vector(7 downto 0);
   CUADRO5 : in std_logic_vector(7 downto 0);
   CUADRO6 : in std_logic_vector(7 downto 0);
   CUADRO7 : in std_logic_vector(7 downto 0);
   CUADRO8 : in std_logic_vector(7 downto 0);
   SALIDAB : out std_logic_vector(7 downto 0); --Salida con el valor de brillo
   SALIDAC : out std_logic_vector(7 downto 0); --Salida con el valor de contraste
   INI : in std_logic; -- Esta señal debe ser colocada a 1 para iniciar el calculo
   TER : out std_logic;-- Esta señal es puesta a '1' cuando se tiene el resultado
   CLKI : in std_logic);   -- Señal de reloj
end UCBC;
```

Figura A.4: Definición de la unidad de cálculo.

Las entradas estan conectadas a un grupo de 8 comparadores, que realizan la comparación de los píxeles vecinos con el pixel central. El resultado de estas comparaciones se encuentran en las señales *INDC(0)* a *INDC(7)*, como se muestra en la figura A.5. En esta misma figura se encuentra el grupo de 8 multiplexores, cuyas salidas *PIXSUP(0)* a *PIXSUP(7)* contienen el valor del pixel vecino cuando es mayor o igual al pixel central, y un valor de cero si es menor al pixel central.

En la figura A.6 se muestra el código usado para realizar la inversion de las señales *INDCX*, esto es con el fin de poder seleccionar los valores de los píxeles menores al valor del pixel central. Esto se logra con el grupo de 8 multiplexores mostrados en la misma figura A.6. De manera semejante al caso anterior, las salidas *PIXINF(0)* a *PIXINF(7)* contienen los valores de los píxeles vecinos si son inferiores al pixel central, y cero en caso contrario.

Después de separar los píxeles vecinos, los mayores o iguales al pixel central de los menores, el siguiente paso es realizar las respectivas sumas. En figura A.7 se muestra como

```
-- Comparación de cada pixel periférico con el pixel central
  C1 : COMP port map (CUADRO0, CUADRO4, INDC(0));
  C2 : COMP port map (CUADRO1, CUADRO4, INDC(1));
  C3 : COMP port map (CUADRO2, CUADRO4, INDC(2));
  C4 : COMP port map (CUADRO3, CUADRO4, INDC(3));
  C6 : COMP port map (CUADRO5, CUADRO4, INDC(4));
  C7 : COMP port map (CUADRO6, CUADRO4, INDC(5));
  C8 : COMP port map (CUADRO7, CUADRO4, INDC(6));
  C9 : COMP port map (CUADRO8, CUADRO4, INDC(7));
-- Selección de los pixels mayores o iguales al pixel central
  MS1 : MUX port map (CUADRO0, INDC(0),PIXSUP(0));
  MS2 : MUX port map (CUADRO1, INDC(1),PIXSUP(1));
  MS3 : MUX port map (CUADRO2, INDC(2),PIXSUP(2));
  MS4 : MUX port map (CUADRO3, INDC(3),PIXSUP(3));
  MS5 : MUX port map (CUADRO5, INDC(4),PIXSUP(4));
  MS6 : MUX port map (CUADRO6, INDC(5),PIXSUP(5));
  MS7 : MUX port map (CUADRO7, INDC(6),PIXSUP(6));
  MS8 : MUX port map (CUADRO8, INDC(7),PIXSUP(7));
```

Figura A.5: Selección de los píxeles mayores al pixel central.

son instanciados dos sumadores para realizar esta labor. Las salidas *TOTS* y *TOTI* son los dividendos de sus respectivos divisores. En la misma figura se muestra el componente *HAMMING*, que nos da el número de píxeles mayores al pixel central (*NUMSUP*), y el número de píxeles menores (*NUMINF*). Estos dos últimos valores son los divisores que hacen falta para obtener los promedios. Estos se encuentran en las señales *PROMSUP* y *PROMINF*. Las señales *RESSUP* y *RESINF* son los residuos, y las señales *TERSUP* y *TERINF* indican la terminación del cálculo de la división. Si se registra una división entre cero, esta es indicada por la señal *INDIV0S* o por *INDIV0I*.

Al valor del promedio es necesario realizar un redondeo, de acuerdo con el valor del residuo. Además, para el caso de que algún divisor detecte una división entre cero, es necesario seleccionar el valor del pixel central para obtener el valor de contraste. Estas labores son realizadas por los módulo A y B que fueron mostrados en la figura 5.18 de la sección 5.2.4. En la figura A.8 se presenta el código usado para implementar el módulo A, que se encarga de definir uno de los valores usados para el cálculo de contraste, el originado por los píxeles vecinos mayores o iguales al pixel central. Para el otro módulo el código es semejante.

Después de checar si hay división entre cero y del redondeo, la señal *CONTSUP* contiene

```
INDCINV(0) <= not INDC(0);
INDCINV(1) <= not INDC(1);
INDCINV(2) <= not INDC(2);
INDCINV(3) <= not INDC(3);
INDCINV(4) <= not INDC(4);
INDCINV(5) <= not INDC(5);
INDCINV(6) <= not INDC(6);
INDCINV(7) <= not INDC(7);
-- Selección de los pixels menores al pixel central
MI1 : MUX port map (CUADRO0, INDCINV(0),PIXINF(0));
MI2 : MUX port map (CUADRO1, INDCINV(1),PIXINF(1));
MI3 : MUX port map (CUADRO2, INDCINV(2),PIXINF(2));
MI4 : MUX port map (CUADRO3, INDCINV(3),PIXINF(3));
MI5 : MUX port map (CUADRO5, INDCINV(4),PIXINF(4));
MI6 : MUX port map (CUADRO6, INDCINV(5),PIXINF(5));
MI7 : MUX port map (CUADRO7, INDCINV(6),PIXINF(6));
MI8 : MUX port map (CUADRO8, INDCINV(7),PIXINF(7));
```

Figura A.6: Selección de los píxeles menores al pixel central.

```
SUM1 : SUMADOR port map (PIXSUP(0), PIXSUP(1), PIXSUP(2), PIXSUP(3), PIXSUP(4),
                         PIXSUP(5), PIXSUP(6), PIXSUP(7), TOTS);
SUM2 : SUMADOR port map (PIXINF(0), PIXINF(1), PIXINF(2), PIXINF(3), PIXINF(4),
                         PIXINF(5), PIXINF(6), PIXINF(7), TOTI);

HAM1 : HAMMING port map (INDC(0),INDC(1),INDC(2),INDC(3),INDC(4),INDC(5),INDC(6),
                         INDC(7),INI,NUMSUP,NUMINF);

D1 : DIVISION port map (TOTS, NUMSUP, INI, TERSUP, INDIV0S, CLKI,
                        RESSUP, PROMSUP);
D2 : DIVISION port map (TOTI, NUMINF, INI, TERINF, INDIV0I, CLKI,
                        RESINF, PROMINF);
```

Figura A.7: Obtención de los promedios.

el valor buscado. La señal *TS* se coloca en 1 para indicar que este valor es válido, momento que es utilizado por la entidad *FFS* para retener el valor obtenido en la señal *CONSUP*, donde permanecerá ahí hasta que sea leida. La señal *VALSUP* indica que el valor está listo. Colocando en 1 la señal *RESET* son puestos a cero tanto *CONSUP* como *VALSUP*, dejando al componente listo para recibir otro dato.

Finalmente, está el módulo *RESTA* mostrado en la figura 5.18, que se encarga de leer los valores finales obtenidos, para realizar la resta y obtener el valor de contraste buscado. En la figura A.9 se muestra el proceso usado para implementar este módulo. Las señales *CONSUP* y *CONINF* de los componentes *FFS* y *FFI* contienen los valores finales. Para realizar la resta, este proceso espera hasta que tanto la señal *VALSUP* como *VALINF*

```
A:process(TERSUP,INDIV0S,PROMSUP,RESSUP,CUADRO4,NUMSUP)
variable CONTRASTE : std_logic_vector(7 downto 0);
variable RESMAY : std_logic_vector(4 downto 0);
begin
  CONTSUP <= "00000000";
  RESMAY := "00000";
  CONTRASTE := "00000000";
  TS <= '0';
  if TERSUP = '1' then
    CONTSUP <= PROMSUP;
    CONTRASTE := PROMSUP;
    RESMAY := RESSUP;
    TS <= '1';
    if INDIV0S = '1' then
      CONTSUP <= CUADRO4;
      CONTRASTE := CUADRO4;
      RESMAY := "00000";
    end if;
    if RESMAY > conv_integer(NUMSUP)/2 then  -- Reliza un redondeo del resultado obtenido, de
      CONTSUP <= CONTRASTE + 1;    -- acuerdo con el residuo
    end if;
  end if;
end process;

FFS : FF port map(CONTSUP,TS,RESET,CONSUP,VALSUP);
```

Figura A.8: Módulo A.

sean 1, en este momento realiza la resta y el valor de contraste es colocado en la señal *SALIDAC*. La señal *TER* es puesta a 1 para indicar que la unidad de cálculo tiene listos los valores de brillo y contratse, correspondientes al pixel central de la ventana de 3x3 seleccionada. Además de esto, este proceso coloca en 1 la señal *RESET* que, como se mencionó, prepara a los componentes *FFS* y *FFI* para recibir otro valor.

```
RESTA:process
begin
  wait until CLKI'event and CLKI = '1';
  if VALSUP = '1' and VALINF = '1' then
    SALIDAC <= CONSUP - CONINF;
    TER <= '1';
    RESET <= '1';
  else
    SALIDAC <= "00000000";
    TER <= '0';
    RESET <= '0';
  end if;
end process;
```

Figura A.9: Módulo A.

Glosario

Código Bayer. Patrón usado por circuitos sensores de imagen. Para obtener la imagen real es necesario aplicar un algoritmo de interpolación.

DRAM: *Dynamic Random Access Memory*, memoria de acceso aleatorio dinámica. Tipo de memoria cuyo contenido debe ser refrescado, o sea, el contenido de las celdas debe ser recargado continuamente.

DSP: *Digital Signal Processor*, procesador de señales digitales. Microprocesador con funciones especiales para el procesamiento de señales digitales.

FPGA: *Field Programable Gate Array*, arreglo de compuertas programables en campo. Circuito integrado de alta densidad que contiene un arreglo de compuertas lógicas cuyas conexiones son programables de tal forma que sea posible introducir sistemas de circuitos lógicos electrónicos en un solo chip para implementar un circuito integrado de propósito específico.

Java: Lenguaje de programación orientado a objetos de propósito general.

Puerto Paralelo. Interface de entrada y salida que usan las computadoras para intercambiar infomación con algún dispositivo externo.

RAM: Random Access Memory, memoria de acceso aleatorio. Conjunto de celdas usado por los sistemas digitales para almacenar información.

SMT: *Surface Mount Technology* tecnología de montaje superficial. Tecnología usada para colocar dispositivos electrónicos sobre tarjetas de circuito impreso, para posteriormente ser soldados. Su característica principal es que no es necesario atravesar la tarjeta

para colocar los dispositivos, como es el caso de la tecnología *through hole*.

SRAM *Static Random Access Memory*, memoria de acceso aleatorio estática. Tipo de memoria cuyo contenido es estático, a diferencia de las DRAM, cuyo contenido debe ser refrescado continuamente para evitar que se pierda.

VHDL: *VHSIC Hardware Description Hardware*, lenguaje de descripción de hardware para VHSIC. Lenguaje usado para diseñar circuitos digitales en dispositivos programables.

VHSIC: *Very High Speed Integrated Circuit*, circuito integrado de muy alta velocidad.

Bibliografía

[1] Tatiana Baidyk, Ernst Kussul, and Oleksandr Makeyev. Texture recognition with random subspace neural classifier. *WSEAS Transactions on Circuits and Systems*, 4(4):319–324, April 2005.

[2] Gonzalez Rafael C. and Woods Richard E. *Digital Image Processing*. Addison-Wesley Publishing Company, United States, 1993.

[3] Ya-Chung Chen and Shu-Yuan Chen. Image classification using color, texture and regions. *Image and Vision Computing*, 21(9):759–776, September 2003.

[4] Allen M. Dewey. *Analysis and design of digital systems with VHDL*. PWS pub., Boston, 1997.

[5] Yu-Chin Hsu. *VHDL modeling for digital design synthesis*. Kluwer, Boston, 1995.

[6] César Torres Huitzil and Miguel Arias Estrada. Real-time image processing with a compact fpga-based systolic architecture. *Real Time Imaging*, 10(3):177–187, June 2004.

[7] EliasÑ. Malamasa, Euripides G.M. Petrakisa, Michalis Zervakisa, and Laurent PetitJean-Didier Legat. A survey on industrial vision systems, applications and tools. *Image and Vision Computing*, 21(2):171–188, February 2003.

[8] Madhav Moganti and Fikret Ercal. Automatic pcb inspection algorithms: A survey. *Computer Vision and Image Understanding*, 63(2):287–313, March 1996.

[9] Eugenia Montiel, Alberto S. Aguado, and Mark S. Nixon. Texture classification via conditional histograms. *Pattern Recognition Letters*, 26(11):1740–1751, August 2005.

[10] P. Paclik, R.P.W. Duin, G.M.P. van Kempen, and R. Kohlus. Supervised segmentation of textures in backscatter images. In *ICPR02*, pages II: 490–493, 2002.

[11] Jianbiao Pan, Gregory L. Tonkay, Robert H. Storer, Ronald M. Sallade, and David J. Leandri. Critical variables of solder paste stencil printing for micro-bga and fine-pitch qfp. *IEEE Transactions on Electronics Packing Manufacturing,*, 27(2):125–132, April 2004.

[12] Ray P. Prasad. *Surface Mount Technology Principles and Practice*. Van Nostrand Reinhold, New York, 1989.

[13] M.J. Swain and D.H. Ballard. Color indexing. *International Journal of Computer Vision*, 7(1):11–32, November 1991.

[14] Synplicity. *Synplify Pro Reference Manual*, June 2003.

[15] Wayne Wolf. *Modern VLSI design System-on-Chip Design*. Prentice Hall, United States, 2002.

[16] Xilinx. *Spartan-3 FPGA Family: Functional description*, August 2004.

[17] Xilinx. *Spartan-3 FPGA Family: Introduction and Ordering Information*, July 2004.

[18] Sudhakar Yalamanchili. *Introductory VHDL from simulation to synthesis*. Prentice Hall, United States, 2001.