UNIVERSIDAD AUTÓNOMA DE QUERÉTARO

FACULTAD DE INFORMÁTICA

DESARROLLO E IMPLEMENTACIÓN DE ALGORITMOS DE PROCESAMIENTO DE IMÁGENES EN DISPOSITIVOS PROGRAMABLES EN CAMPO FPGA

Tesis

Que como parte de los requisitos para obtener el
Título de

INGENIERO DE INCOMPUTACIÓN

PRESENTA

CARLOS ALBERTO RAMOS ARREGUÍN

DIRIGIDO POR:

Dr. JESÚS CARLOS PEDRAZA ORTEGA

Santiago de Querétaro, Qro, México, Septiembre 2010

Universidad Autónoma de Querétaro
Facultad de Informática
Ingeniería en Computación
Plan INC 99

DESARROLLO E IMPLEMENTACIÓN DE ALGORITMOS DE PROCESAMIENTO DE IMÁGENES EN DISPOSITIVOS PROGRAMABLES EN CAMPO FPGA

TESIS

Que como parte de los requisitos para obtener el Título de
Ingeniero en Computación

Presenta:
CARLOS ALBERTO RAMOS ARREGUÍN

Dirigido por:
Dr. Jesús Carlos Pedraza Ortega

SINODALES

Dr. Jesús Carlos Pedraza Ortega Presidente	_____ Firma
Dr. Marco Antonio Aceves Fernández Secretario	_____ Firma
Dr. Saúl Tovar Arriaga Vocal	_____ Firma
Dr. José Emilio Vargas Soto Suplente	_____ Firma
Dr. Efrén Gorrostieta Hurtado Suplente	_____ Firma

_____
M. C. Ruth Angélica Rico Hernández
Directora de la Facultad de Informática

Centro Universitario
Querétaro, Qro.
Septiembre 2010 México

Este trabajo es dedicado a la memoria de mi abuelita Manuela Rodríguez, en paz descanse.

A mis padres Ma. de la Luz y Juan, por el apoyo que siempre me ofrecieron y su confianza para haber logrado este objetivo. A mis hermanos Enrique y Marco que a pesar de nuestras diferenciasme ha brindado su apoyo durante mi crecimiento como persona. A Juan Manuel por todo su apoyo que me ha brindado no solo como hermano, sino también, en toda mi formación académica. A mis cuñadas Sofía Mayela y Maricela, por sus consejos en algunas etapas de mi vida. A mis sobrinos Enrique Iván, Alexis Alejandro, Angélica Abigail y Maricela Eliana, por formar parte de esta familia. A toda la familia Ramos y la familia Arreguín, en especial a mi madrina María Elena.

A mis amigos de la infancia Luis Ángel, en paz descanse, Javier y Juan Alberto, que a pesar, de caminos distintos me han apoyado cuando los he necesitado. A Ma. Guadalupe, Juan Carlos, Jorge Martín, Hernán, por permitirme ser su amigo y su apoyo a lo largo de mi carrera profesional. A Mario y Alejandro, que a pesar del corto tiempo de conocernos, me han brindado su apoyo y consejos.

A todos mis maestros, por sus enseñanzas, y a la directora M. C. Ruth Angélica Rico, por permitirme formar parte de CIDIT de la Facultad de Informática. A los doctores Efrén Gorrostieta, Marco Antonio Aceves, Sandra Luz Canchola y mi asesor Jesús Carlos Pedraza, por su apoyo para la realización de este proyecto y sus consejos brindados a mi persona.

Al proyecto

A todos ustedes, Gracias.

En este trabajo se propone un sistema básico de procesamiento de imágenes basado en FPGA con implementación de transformación de imágenes. Se desarrolla un controlador de sincronía VGA para visualizar imágenes en un monitor de tipo CRT, para realizar esto se describen dos metodologías.

Se proponen dos metodologías para la visualización de imágenes con FPGA, de las cuáles, debido a la eficiencia que se obtuvo, esta eficiencia se mide en el consumo de recursos del FPGA, obteniendo una velocidad de procesamiento mejor que el de una PC convencional.

El lenguaje descriptivo utilizado es VHDL, utilizando librerías estándar del IEEE, para conservar la portabilidad del diseño. En la parte inicial del proyecto, que corresponde a la sincronía con el monitor, se utilizaron tres tarjetas, para probar el desempeño de cada una en la exhibición de imágenes. Como resultado, se tiene un sistema básico para realizar procesamiento de imágenes en hardware, donde se aplican tres transformaciones a la imagen que son: negativo de color, escala de grises y binarización por umbral. Finalmente, se presenta un método para la implementación de filtros.

ÍNDICE

CAPÍTULO I
INTRODUCCIÓN

Desde 1964 hasta nuestros días, el campo de procesamiento de imágenes ha crecido enormemente. Las técnicas de procesamiento se usan ahora para resolver una gran variedad de problemas. Aunque a menudo no relacionados, esos problemas requieren comúnmente métodos capaces de realizar y extraer la información contenida en las imágenes para su interpretación y análisis por parte de los humanos. En cualquier caso se contemplan tanto técnicas de mejora de la calidad de las imágenes como relativas a la percepción de la máquina.

El creciente desarrollo de las tecnologías y, en especial de la información hace que cada día sea más frecuente su uso en el tratamiento de imágenes.

Por otra parte, el desarrollo de equipos cada vez más sofisticados en diversos campos de aplicación de las imágenes constituyen la base hace que el tratamiento de las imágenes sea algo inherente al desarrollo tecnológico. A modo de ejemplo, citemos dos casos de especial relevancia:

a) *Medicina,* donde los sofisticados equipos PET (***Positrón Emisión Tomography***), resonancia magnética, rayos X, etc., se verían mermados en su potencialidad de diagnosis de no ser por un tratamiento apropiado de las imágenes digitales que generan.

b) *Observación de la Tierra,* donde los sensores acoplados en los satélites artificiales son capaces de proporcionar imágenes en las que se pueden observar detalles de hasta0.6 metros de tamaño [1].

c) *Reconstrucción de Objetos 3D,* es el proceso mediante el cual es posible duplicar las características físicas de un objeto obteniendo una así una réplica de la original, existen gran variedad de métodos y técnicas que se utilizan para este fin, una de ellas es a través del procesamiento de imágenes, donde se digitaliza el objeto y posteriormente se realiza su reconstrucción. El uso del procesamiento de imágenes en la reconstrucción 3D es relativamente nuevo pero su investigación ha avanzado en los últimos años. [2-5].

Gracias al tratamiento de las imágenes que generan es posible detectar zonas deforestadas, evolución de fenómenos meteorológicos, etc.

Aunque la distinción entre procesamiento y análisis de imágenes no es obvia de forma inmediata, el procesamiento de imágenes puede ser visto como una transformación de una imagen en otra imagen, es decir, a partir de una imagen, se obtiene otra imagen modificada. Por otro lado, el análisis es una transformación de una imagen en algo distinto a una imagen, en consecuencia, el análisis de imágenes es un determinado tipo de información representando una descripción o una decisión. En cualquier caso, las técnicas de análisis de imágenes digitales son aplicadas a imágenes que han sido procesadas previamente.

1.1. ANTECEDENTES

En 2005 Toledo propone un Sistema Generador Xilinx (XSG), el cual, puede ser usado para desarrollar algoritmos de visión por computadora basados en hardware, también demostrando que Simulink puede ser utilizado como una plataforma de co-diseño y co-simulación para prototipos de sistemas de visión por computadora HW/SW. Para realizarlo, componentes de una librería de procesamiento de imágenes optimizados, basados en XSG y Matlab tienen que ser desarrollados y probados en esquemas híbridos incluyendo módulos de Hardware y Software. Como parte de las pruebas, el sistema fue utilizado para el monitoreo de segmentos de Tangerina [7].

Así mismo en 2005 Sánchez, diseña un circuito comparador de imágenes basado en FPGA, para analizar imágenes de circuitos por textura implementada en hardware con pastas depositadas, 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 una tarjeta sin defectos [8].

En 2006 Quintero propone las metodologías empleadas en un FPGA para realizar el procesamiento de imágenes y el reconocimiento de objetos dentro de dichas imágenes, se encontró un alto desempeño medido en cuadros por segundo, evitando procesamientos

matemáticos complejos. Como resultado final se obtuvo un procesador de imágenes con una limitante de algoritmos iterativos de etiquetado e identificación de objetos [9].

En 2007 Bravo, propone una arquitectura basada en FPGA para la detección de objetos en movimiento utilizando visión computacional y técnicas PCA, logrando un tiempo de ejecución inferior al de una PC de altas prestaciones. Así mismo, esta alternativa aporta una facilidad para la expansión del cálculo de autovalores y autovectores a matrices de diferentes dimensiones, sin apenas incrementar el número de recursos internos del FPGA [10].

En los trabajos anteriormente mencionados, se ha observado que se utilizan librerías no estándar, que son del fabricante del FPGA o herramientas del mismo, por ejemplo: Xilinx. En este trabajo se propone trabajar únicamente con los estándares IEEE desarrollando herramientas propias, por ejemplo; la transferencia de los datos de la imagen al FPGA o a una memoria de tipo SDRAM, para trabajar con FPGAs de diversos fabricantes como: Altera, Xilinx, Cypress. Y, así no llevar acabo adaptaciones por cada arquitectura, con excepción de las terminales.

1.2. OBJETIVOS

1.2.1. General

El objetivo principal de este trabajo es desarrollar una arquitectura propia, para llevar a cabo procesamiento de imágenes en sistemas embebidos utilizando FPGA, únicamente utilizando los estándares del IEEE.

1.2.2. Específicos

1. Comprender el funcionamiento de sincronía de un monitor VGA y diseñar una descripción de hardware para el manejo de monitores tipo CRT.
2. Diseñar una descripción en hardware para la generación de imágenes en monitores CRT.
3. Diseñar un algoritmo para cargar imágenes en FPGA y puedan ser mostradas en monitores estándar.
4. Implementar diversos algoritmos para tratamiento de imágenes, utilizando el lenguaje descriptivo de hardware.

1.3. JUSTIFICACIÓN

La rápida evolución de la tecnología de imágenes digitales, acompañado por una gran demanda en el mercado de cámaras y pantallas, presentan un significante cambio para los desarrolladores de dispositivos, quienes desean crear productos de alta calidad.

Sofisticados algoritmos de procesamiento de imágenes están disponibles, para llevar a cabo una extracción de imágenes del hardware de captura y exhibición. Pero su implementación está limitada por algunos factores como: la complejidad intrínseca del algoritmo, la presión para reducir el costo de los materiales, la necesidad para soportar una

amplia variedad de formatos, y el frecuente requisito para personalizar un ambiente particular de los dispositivos [6].

Hoy en día la clave para la implementación de sistemas de procesamiento de señales digitales (DSP), especialmente en comunicaciones digitales, aplicaciones de procesamiento de imágenes y video, es el uso de dispositivos lógicos programables, en particular FPGA (Field Programmable Gate Arrays) [7], el cual en la actualidad, ha comenzado a ser muy utilizado en el área de software embebido, ya que son tan eficientes como un ASIC (Application Specific Integrated Circuit) y compiten en costo con los microcontroladores [9].

Usualmente, los FPGAs son programados utilizando lenguajes de descripción de hardware. En nuestro caso se trabaja con el lenguaje descriptivo VHDL (Very high speed integrated circuit Hardware Description Language), desconocido para muchos desarrolladores de sistemas embebidos, por lo cual prefieren lenguajes de alto nivel, para un rápido desarrollo de alguna aplicación deseada [11].

La ventaja de utilizar FPGA en el trabajo, nos permite crear una arquitectura propia, la cual podrá ser utilizada en descripciones más complejas sin importar el fabricante de FPGA. Así mismo, debido a la diversidad de trabajo en el procesamiento de imágenes, dónde no será necesario utilizar una computadora convencional de propósito general, y al tener un hardware que tenga un propósito específico, nos ayuda en ahorro de costos y podremos obtener una eficiencia mayor por parte del sistema.

Para este trabajo se utilizaron 3 tarjetas de desarrollo con FPGA, en la sección 3.3, se explica el puerto VGA de cada una de ellas. Para comenzar, es conveniente conocer cómo realizar la sincronía entre el hardware (FPGA y Monitor VGA), lo cual se describe en el capítulo 3 de este trabajo. Pero antes, es importante conocer acerca de los **Sistemas Embebidos**, descrito en el capítulo 2.

1.4. ORGANIZACIÓN DE LA TESIS

La organización de este trabajo se presenta en 6 capítulos, desglosando cada parte importante con su teoría y fundamentos para llevar a cabo este proyecto de investigación.

En el capítulo 1 se presentan las generalidades y direcciones de la tesis, donde se presenta una introducción para tener una idea general del proyecto y conocer las consideraciones que se tomaron en cuenta para la realización del mismo.

En el capítulo 2 se presentan bases de un sistema embebido, ya que es el área de especialidad del proyecto, explicando a detalle las partes fundamentales que componen un sistema embebido.

En el capítulo 3 se presentan las bases para lograr una correcta sincronización con un monitor VGA CRT, trabajando una resolución de 680x480 pixeles. Así mismo, se define una tarjeta de desarrollo a emplear en la parte fuerte del proyecto. Y en este capítulo se define el tipo de tarjeta utilizada para las pruebas finales y más importantes de este trabajo.

En el capítulo 4 se presentan los fundamentos de procesamiento digital de imágenes, dando una breve introducción. Presentando los fundamentos de las técnicas empleadas para obtener un negativo a color de la imagen original, la conversión de la imagen de color a escala de grises y binarización por umbral.

En el capítulo 5 se presenta la implementación de la arquitectura utilizada, para lograr almacenar una imagen en FPGA y ser mostrada en el monitor. Se presentan dos métodos: el primero es creando una tabla de datos que es almacenada en el FPGA, y el segundo utilizando una memoria de tipo SDRAM para el almacenamiento de los datos de la imagen.

En el capítulo 6 se presentan las pruebas y resultados obtenidos con la arquitectura desarrollada para el FPGA seleccionado, y comparaciones de los resultados obtenidos en nuestra plataforma contra los resultados obtenidos en una PC, utilizando MATLAB.

CAPÍTULO II
SISTEMAS EMBEBIDOS

2.1. INTRODUCCIÓN [12]

Se conoce como ***sistema embebido*** a un circuito electrónico computarizado que está diseñado para cumplir una labor específica en un producto.

La inteligencia artificial, secuencias y algoritmos de un sistema embebido, están residentes en la memoria de una pequeña computadora denominada microcontrolador o dispositivo lógico programable.

A diferencia de los sistemas computacionales de oficina y *laptops*, estos sistemas solucionan un problema específico y están dispersos en todos los ambientes posibles de la vida cotidiana. Es común encontrar sistemas embebidos en vehículos, controlando el sistema de inyección de combustible, sistemas de frenado antibloqueo, control de espejos, sistemas de protección contra impacto (bolsas de aire), alarmas contra robo, sistema de encendido, sistemas de ubicación, etc. Así mismo en lavadoras, estufas, refrigeradores, hornos de microondas; equipos celulares, agendas de bolsillo, PDA, cajeros automáticos, cámaras fotográficas, etc. En consecuencia, un sistema embebido lo encontramos en muchos de los aparatos que utilizamos en nuestra vida cotidiana.

En la actualidad un consumidor promedio interactúa con alrededor de 400 sistemas embebidos por día, lo cual tiende a crecer significativamente para los próximos años, considerando que el hardware es cada vez más pequeño, consumen menos energía y el costo es menor gracias a la economía de escala aplicada en la fabricación, aspectos que ayudan a reemplazar en mayor proporción los sistemas lógicos, los equipos electromecánicos y en el futuro, se podrán incorporar en los equipos desechables.

Los elementos principales de un sistema embebido son los siguientes:

a. Software
b. Hardware

Estos dos elementos se describen más a detalle en las secciones 2.3 y 2.4.

2.2. SISTEMAS EMBEBIDOS [13]

Los sistemas embebidos generalmente son usados para una aplicación específica. Varias características funcionales asociadas con sistemas embebidos son:

- Bajo costo
- Pocos componentes
- Consumo de energía bajo
- Proporcionan respuesta y ayuda en tiempo real de la coexistencia de equipo y programas.

Una metodología general empleada en el diseño de sistemas embebidos es mostrada en la *figura 2.1*.

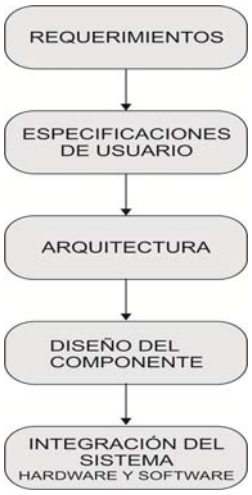


Figura 2.1. Metodología general empleada para Diseño de Sistemas Embebidos

La metodología anterior se explica de la siguiente manera:

- ***Requerimientos***: Se refiere a todo requerimiento funcional y no funcional, como lo pueden ser: tamaño, peso, consumo de energía y costo.
- ***Especificaciones de Usuario***: Detalles de la interfaz de usuario junto con operaciones necesarias para satisfacer la petición de usuario.

- ***Arquitectura***: Hardware (procesador, periféricos, lógica programable y ASSPs) y Software (programas de mando y sus operaciones).

- ***Diseño del Componente***: Componentes prediseñados, componentes modificados y nuevos componentes.

- ***Integración del Sistema***: Esquema de verificación para encontrar errores rápidamente.

La decisión de la clase de plataforma digital a utilizar, se toma durante la fase de diseño de la arquitectura del sistema, debido a que cada aplicación embebida tiene sus propias necesidades de acuerdo a los puertos operacionales que requiera. Algunos de los puertos de un sistema embebido en hardware incluyen (no particularmente en orden) lo siguiente:

- Actualización en tiempo real
- Energía
- Costo
- Encapsulado
- Facilidad de programación
- Portabilidad del código
- Librerías de código reutilizable
- Herramientas de programación

2.2.1. Plataformas de Diseño Digital

Hasta la década de los 70’s, los diseños de sistemas electrónicos fueron basados en componentes analógicos discretos tales como: transistores, amplificadores operacionales, resistores, condensadores e inductores. Estos circuitos ofrecieron procesamiento concurrente, pero tenían problemas con parámetros de temperatura y tiempo de vida. Los

componentes basados en TTL iniciaron el diseño digital. En 1971, el microprocesador de Intel 4004, se convirtió en la primera plataforma digital configurable mediante software en el año 1971. La *tabla 2.1* enlista los diseños digitales contemporáneos principales junto con su mérito.

Tabla 2.1. Plataformas de Diseño Digital

Plataforma de Diseño Digital	Merito	Año
Microprocesadores	Reconfigurable utilizando software. bueno para computadoras	1971
Microcontroladores, Controladores de Señales Digitales	Combinación de periféricos y CPU	1976
Producto Estándar de Aplicación Específica (ASSP)	Un periférico especializado con la habilidad de comunicar con un procesador principal	1984
Arreglo de Compuertas Programables en Campo (FPGA)	Habilidad de combinar los puntos fuertes del procesador, controlador y ASSP	1988

2.2.2. Diseño Basado en Microprocesadores

El microprocesador ha cambiado la metodología del diseño digital como ningún otro componente digital. Comenzó como un CPU de 16 bit en 1971, y continúa siendo el controlador digital para varias áreas de aplicación. Éste trajo el concepto de la arquitectura sistema de instrucción (ISA), ensamblador y compilador. Hay muchas aplicaciones en tiempo real, con frecuencias de actualización rápidas que requiere su tradicional lenguaje ensamblador. Esto usualmente es hecho cuando el tamaño de la memoria disponible es una restricción. Aunque la mayoría de los microprocesadores empleados hoy en día abastecen aplicaciones de datos céntricos, hay núcleos de microprocesadores embebidos en microcontroladores para aplicaciones de control en tiempo real.

2.2.3. Diseño Basado en Microcontroladores

Los microcontroladores representan la próxima generación de controles para sistemas embebidos. Debido a que permiten crear sistemas con un número de componentes reducido, y además se han incorporado periféricos que anteriormente eran interconectados externamente con el procesador de propósito general.

Como en el microprocesador, el ambiente de diseño de las tareas en un microcontrolador, son divididos según la frecuencia de actualización requerida. Para tareas que requieren una frecuencia de actualización baja, el código es completado usando un software de programación como el lenguaje C. Las tareas que necesita tener una actualización determinada, con una frecuencia alta, es codificado usando el lenguaje nativo ensamblador para un microcontrolador en particular.

2.2.4. Productos Estándar de Aplicación Específica

Es un componente lógico configurable para una aplicación específica. La funcionalidad de un ASSP está basada especificando su palabra de control. Los ASSPs son hechos en grandes volúmenes y abastecen los requerimientos genéricos de una aplicación. Los diseños basados en ASSP son usados en un PCB (Printed Circuit Board) para que un producto pueda salir al mercado. En la aplicación de un robot, puede ser utilizado para controlar el motor de cada eje, por ejemplo.

2.2.5. Diseño Empleando FPGA

Hoy en día el FPGA proporciona una plataforma que soporta peticiones de procesamiento y lógica personalizada. Actualmente, los microcontroladores tienen una ventaja sobre los FPGA en términos de consumo de energía y costo. Pero los FPGAs los están alcanzando ofreciendo portabilidad del código a través de varios fabricantes de FPGA, bibliotecas de código reutilizables y disponibilidad de herramientas de programación baratas. Los dispositivos programables que tradicionalmente se basan en compuertas lógicas sencillas, ahora están en una posición para dar soporte a grandes partes de un sistema digital. Hoy en día el diseñador digital tiene una opción viable de usar solamente el FPGA como el controlador del sistema embebido. La disponibilidad de los dispositivos FPGA de alta densidad ha dado a los diseñadores digitales lotes de flexibilidad para diseñar arquitecturas digitales personalizadas usando FPGA y Lenguajes Descriptivos de Hardware (HDL). Estos dispositivos contienen una gran variedad de componentes

digitales incorporados (memoria, multiplicadores, transmisores-receptores y más), la densidad de este dispositivo se ha incrementado con el paso del tiempo, al igual que su costo, y se ha hecho económicamente viable para diversas aplicaciones. El FPGA contemporáneo contiene millares de Tablas de acceso a datos (LUT, Look Up Tables) y elementos de memoria (FF, Flip- Flops) para la implementación de un sistema digital complejo.

Los FPGAs contemporáneos ofrecen:

- ***Reconfigurabilidad***: Los dispositivos programables de campo pueden ser reconfigurados en cualquier momento, lo que permite al diseñador integrar en cualquier momento modificaciones o hacer cambios personalmente.

- ***Diseño de Software definido***: El hardware es definido por el software de lenguajes HDL. Los diseñadores pueden desarrollar, simular y probar un circuito completamente antes de implementarlo en un FPGA.

- ***Paralelismo***: Los circuitos definidos en un FPGA pueden ser diseñados en una forma completamente paralela. Esto es, similar a usar los dispositivos análogos de trayectoria múltiple. Un usuario puede realizar múltiples implementaciones en hardware en el mismo chip sin interferencia por cruce de módulos o cargamento de cómputo. Un ejemplo de procesamiento concurrente basado en FPGA es mostrado en la *figura 2.2.*

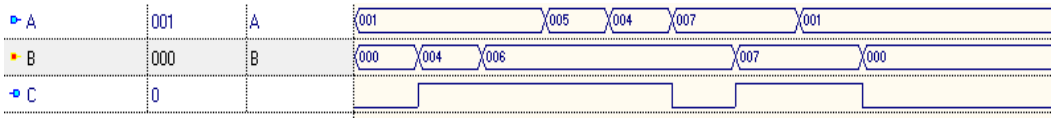


Figura 2.2. Procesamiento concurrente basado en FPGA

- ***Alta Velocidad***: Un FPGA es un dispositivo que funciona con frecuencias de reloj altas. El diseñador puede alcanzar altas velocidades. Acoplado con paralelismo, la implementación en FPGA puede superar sistemas basados en procesadores.

- ***Confiabilidad***: los diseñadores pueden contar con mucha confiabilidad del hardware de FPGA porque no hay sistema operativo o driver de capa 3que pueda afectar el tiempo del sistema.

- ***Protección y reutilización IP***: Una vez compilado y transferido al FPGA, la ingeniería en reversa de la implementación de hardware es complicada de realizar.

Los sistemas basados en FPGA han ganado aceptación porque estos sistemas integran diseño en lógica digital, interfaces de procesadores y comunicaciones en un solo chip. El diseño del flujo de las partes frontales es muy similar a un diseño lógico personalizado. Casi todos los fabricantes de FPGA ofrecen un software con herramientas que permiten a un diseñador, simular, sintetizar, colocar y direccionar, y programar el FPGA. La *tabla 2.2* muestra las diferentes herramientas de diseño ofrecidas por cuatro principales fabricantes. Una vez que un diseñador se siente familiarizado con un software en particular, es fácil migrar a herramientas de diseño de otro fabricante porque trabajan de forma similar. El único "inconveniente", es conocer las herramientas que proporciona cada fabricante para la interfaz de programación de los dispositivos.

Tabla 2.2. Herramientas de diseño de cuatro fabricantes de FPGA

FUNCIONALIDAD	XILINX	ALTERA	ACTEL	LATTICE
Síntesis de diseño, mapeo, lugar y direccionar.	Integrated Software Environment (ISE)™	Quartus II®	Libero® IDE	ispLEVER
FPGA herramienta del procesador embebido.	Embedded Design Kit (EDK)®	System on Programmable Chip (SoPC) builder®	SmartDesign	
Soporte de personalización de periféricos	Si	Si	Si	Si
Analizador de señales lógicas en el Chip	ChipScopet™ Pro	Signal Tap®		
Librerías MATLAB® cosimulación y núcleos IP	System Generator™	DSP Builder®		

2.3. SOFTWARE EMBEBIDO [14]

Los sistemas embebidos están por todas partes. No puedes alejarte de ellos. En casi todos los hogares de América, hay alrededor de 40 microprocesadores sin contar las PCs (las cuáles contribuyen entre 5 a 10 más) o automóviles (los cuáles contienen al menos una docena). Y estos números son parecidos a incrementarse en una magnitud superior a la actual en una década o dos. Es algo irónico que mucha gente que está fuera del negocio de la electrónica, no posea idea alguna de lo que realmente significa “embebido”.

La gente del comercio aprecia mucho el mercado de segmentos. La teoría es que tal segmentación analiza los requerimientos completamente de un producto de cada segmento en una forma específica. Para embebido, tenemos mercados como telecomunicaciones, control aéreo y procesos, consumibles y automóviles. Aunque cada vez más, los dispositivos abarcan más y más áreas que no entran en este modelo. Por ejemplo, un teléfono celular con cámara, ¿es un producto de telecomunicaciones o consumible?, ¿A quién le importa? Una interesante área de consideración es la concordancia de tales aplicaciones, un comentario de ello es la cantidad de software está creciendo fuera de todo reconocimiento.

2.3.1. Retos de Desarrollo

Las aplicaciones consumistas están restringidas por el mercado y su costo. Esto lleva a algunos desafíos interesantes en el desarrollo de software.

2.3.2. Múltiples procesadores

Los diseños de sistemas embebidos incluyen más de un procesador en común. El mayor reto con múltiples procesadores es la depuración. El código en cada dispositivo individual puede ser depurado, si las herramientas y técnicas son bien entendidas. El reto se presenta con las interacciones entre dos o más procesadores. Hay una necesidad clara de eliminar errores en la tecnología empleada, esto es depuración.

2.3.3. Memoria Limitada

Los sistemas embebidos casi siempre tienen memoria limitada. Aunque la cantidad de memoria puede ser no pequeña, típicamente no puede ser agregada sobre demanda. Para cualquier aplicación, una combinación de costo y consumo de energía estas consideraciones pueden resultar que la cantidad de memoria también sea una restricción. Tradicionalmente, los ingenieros de software embebido han desarrollado habilidades programando en un ambiente disponible con memoria limitada.

2.3.4. Interface de usuario

La interfaz de usuario en cualquier dispositivo es críticamente importante. Su calidad puede tener una influencia directa en el éxito de un producto. Con un producto de consumo, la influencia es determinante, si los usuarios creen que la interface es “anticuada” o tiene fallas, la percepción del dispositivo en particular no sólo será afectada, sino la marca entera. Cuando es hora de mejorar el usuario buscará otras opciones.

En general, la interfaz de usuario no se ejecuta en hardware. Las funciones de varios controles, son definidos por software, y puede haber muchos controles o muy pocos. Por lo tanto, en un mundo ideal, la secuencia del desarrollo sería:

- Diseñar el hardware
- Hacer prototipos
- Implementar el software (interfaz de usuario)
- Probar el dispositivo con la interfaz de usuario y modificar cuando sea necesario

Pero no vivimos en un mundo ideal…

En el mundo real, la complejidad del software y las restricciones del mercado demandan que el software sea completado antes de que el hardware éste disponible. De hecho, mucho del trabajo típicamente necesita ser hecho antes de que se termine el diseño

del hardware. Una aproximación a este dilema es el uso de tecnología prototipo. Con tecnología moderna se puede realizar la simulación, se puede probar el código, junto cualquier sistema operativo, en nuestra computadora de desarrollo (Windows, Linux, Unix, etc…) y se liga a una representación gráfica de la Interfaz de Usuario. Esto permite a los desarrolladores interactuar con el software como si se tuviera el dispositivo en manos propias.

2.3.5. Software Reutilizable

Comparado con la programación convencional en computadora, restringida por el sistema operativo y por algún otro software, programar un sistema embebido es parecido al trabajo en un ambiente donde el desarrollador podría tener el control total. Pero las cosas han cambiado, las aplicaciones son grandes y complejas que es usual que ingenieros de software estén involucrados en un equipo de trabajo. El tamaño de la aplicación significa que un individuo nunca podría completar el trabajo en tiempo; la complejidad significa que pocos ingenieros tendrían la habilidad de completar el sistema. Con tiempos cortos el mercado incrementa, es ahí donde se encuentra un incentivo para reutilizar el código existente, desde dentro de la compañía o si es permitido por fuera de la misma.

El reutilizar los diseños de propiedad intelectual en general, es común y bien aceptado en el mundo del diseño de hardware. Para el software de escritorio, es ahora una estrategia de implementación común. Los ingenieros de software embebido tienden a ser conservadores y no adoptar nuevas ideas, pero esta tendencia necesita cambiar.

2.3.6. Componentes de Software

Es entendible y esencial que hay que incrementar el reuso del código, pero amerita una revisión de las posibilidades.

A continuación se mencionan algunos de los componentes claves que pueden ser autorizados y considerados las cuestiones claves

2.3.7. Sistema Operativo en Tiempo Real (RTOS)

El tratamiento de un sistema operativo en tiempo real como componente de software no es nuevo, hay alrededor de 200 productos en el mercado. La diferencia algunas veces es clara, pero en otros casos es más sutil. Se puede aprender mucho de los criterios de selección para un sistema operativo de tiempo real.

2.3.8. Factores de Selección de un Sistema Operativo en Tiempo Real

El estudiado detallado de mercados ha revelado algunas tendencias en los factores que conducen a las decisiones de compra de productos de sistemas operativos en tiempo real.

- **Tiempo Real Duro**: “Tiempo Real” no necesariamente significa “rápido”, significa “bastante rápido”. Un sistema en tiempo real es, sobretodo, predecible y determinista.

- **Derechos Libres**: La idea de alguna licencia de software, y pagar algo para su uso, puede ser poco atractivo. Para volúmenes más grandes, en particular, un modelo de derechos libres es ideal. Un modelo comercial flexible, reconoce que todos los sistemas embebidos son diferentes, es el requisito.

- **Soporte**: Un sistema operativo en tiempo real es un producto altamente sofisticado. La disponibilidad de soporte técnico de alta calidad no es una opción.

- **Herramientas**: Un fabricante RTOS puede referirle por otra parte las herramientas o revender simplemente algún producto de otra compañía. Esta práctica no disminuirá el nivel de integración de herramientas para RTOS requeridos para el desarrollo de un sistema eficiente.

- **Fácil de Usar**: Como un factor de selección, el uso fácil en un RTOS lo hace atractivo. En realidad, programar un sistema de tiempo real, no es fácil; requiere de un gran esfuerzo. El fabricante puede ayudar suministrando código fuente comentado, cuidadosamente integrado junto con los componentes del sistema.

- **Networking**: Aproximadamente una tercera parte de todos los sistemas embebidos están conectados, esto se convierte en un requisito común.

- **Soporte Amplio**: El soporte, dado por una arquitectura RTOS, de una amplia gama de microprocesadores es un beneficio obligatorio.

2.3.9. Memoria en Sistemas Embebidos

Cuando la gente discute los avances en la microelectrónica durante los últimos años, el uso de la memoria se ha incrementado y por lo tanto va mejorando con el paso del tiempo. Un ejemplo de ello, lo tenemos como herramienta en nuestra vida cotidiana la PC, que tiene unos 20 años de edad; el modelo original tenía solo 16K de memoria, mientras que 512M se considera ahora una cantidad media. Antes de comenzar a detalle con la sección de memoria veamos 3 conceptos de distinto punto de vista.

Es un chip en el cual se pueden mantener bits de datos. Hay dos tipos de memoria: ROM y RAM. Estos vienen en dos variedades cada uno. ROM programada y dispositivos programables, el cual puede programar uno mismo. La RAM puede ser estática, la cual es fácil de usar pero tiene menor capacidad; la dinámica es más densa pero necesita circuitos de soporte. Este concepto se refiere a memoria como hardware.

Un concepto por parte de ingeniería de software sería: La memoria es donde funciona un programa se leen los datos y el código dentro de la memoria del disco, y el programa se ejecuta. No es necesario preocuparse tanto por el tamaño, ya que la memoria virtual es efectivamente ilimitada.

Por el lado de Sistemas Embebidos la memoria viene en dos variedades, que son:

- ROM: donde se almacenan el código y las constantes.
- RAM: donde se mantienen los datos variables, pero esta contienen puros datos inservibles al inicio.

2.3.10. Habilitar código ROM

El primer reto y más obvio cuando se implementa software para un sistema embebido es arreglar el código para ser almacenado en ROM y datos variables a ser asignados en espacio RAM. Esto se diferencia de una computara “normal”, donde el código y los datos se cargan simplemente en memoria como unidad. Por lo tanto, los datos pueden ser mezclados con el código y sus valores se pueden fijar desde la compilación. Además, aunque no es una práctica recomendada, los programas pueden ser modificados desde el código almacenado en la memoria. Un compilador cruzado genera código que es habilitado como ROM. En adición, un link para cada aplicación que facilite el posicionamiento entre el código de la ROM y los datos de la RAM.

2.3.11. Secciones de Programa

Para acomodar la necesidad de tratar diferentes tipos de memoria, se desarrolla el concepto de “sección de programa”. La idea es que la memoria se podría dividir en un número de unidades nombradas, llamadas **secciones**. Mientras que la codificación en ensamblador, el programador podría especificar, las secciones donde se coloca el código, los datos, las constantes, etc.

La asignación real de las direcciones de memoria a las secciones de programa ocurre en el tiempo de acoplamiento. El link es proporcionado con direcciones iniciales para cada sección o donde inician las direcciones en una secuencia de secciones, puede ser colocado

en un orden específico. Las contribuciones de un número de objetos en un módulo se pueden hacer a una sección dada. Normalmente, estos se concatenan y se colocan en una dirección especificada.

Para un sistema embebido, en el nivel más simple, apenas dos secciones de programa son necesarias, uno para el código y las constantes (ROM) y uno para los datos (RAM). En cada módulo, el programador tiene cuidado al indicar la sección apropiada para cada parte del programa. En el tiempo de acoplamiento, todo el código y constantes se recolectan juntos y se ponen en la ROM, y todos los datos se ponen en RAM.

En C, cualquier variable que no es automática, es almacenada estáticamente. Una variable estática tiene memoria asignada en el tiempo de compilación. Cuando tal variable se declara es mejor proporcionar un valor inicial, en caso de que no se especifique ningún valor la variable se fija en 0.

Claramente hay un problema potencial con el uso de variables estáticas en un sistema embebido. La disposición de los valores en tiempo de compilación sería perdida, pues solamente el código (y los datos constantes están dentro de la ROM. Hay tres soluciones posibles:

1. No usar variables estáticas inicializadas, utilice asignaciones específicas para fijar sus valores y nunca asumir que una variable no asignada contiene cero. Aunque este acercamiento es posible es inconveniente.

2. Trace las variables estáticas inicializadas dentro de la ROM. Esto quiere decir que aunque tengan el valor inicial requerido, no pueden ser cambiados en lo absoluto más adelante. Mientras que esto puede parecer una restricción, a menudo es útil tener LUTs de variables, que se tratan realmente como constantes.

3. Trace las variables a una RAM, si lo permite el paquete del compilador en uso, con sus valores iniciales que son mapeados a la ROM. Es al fácil copiar un área de la memoria (ROM) a otra (RAM) en el arranque, antes de que la parte principal sea llamada.

2.3.12. Cuando Sale Todo Mal

Al desarrollar el software para un dispositivo embebido, la reconciliación de los varios tipos de memoria vistos, puede no ser fácil. Si el compilador divide simplemente los datos en dos secciones destinadas para la ROM y la RAM respectivamente, las posibilidades de resolver el problema son limitadas. Además, el compilador puede tratar secuencias de datos, que sería incómodo porque necesitan estar en la ROM con el código.

2.3.13. Cuando Sale Todo Bien

Una verdadera herramienta en un sistema embebido puede ser el uso de memoria de manera directa. El uso de las secciones de programa se emplea al máximo. Esta ayuda comienza con el compilador, que genera un número de secciones.

2.3.14. Arquitecturas de Memoria

Hoy en día, todo parece tener una arquitectura, un chip, es un buen ejemplo, donde se tiene el término “propiedades reales de silicio”, que lleva el sector análogo con la construcción de la industria un poco al futuro.

2.3.15. Las Opciones

En general, la arquitectura de la memoria entra en cinco categorías:

1. ***Solo espacio plano***

   La memoria plana es el concepto de arquitectura más fácil de apreciar. Cada posición de memoria tiene una sola dirección, y cada dirección se refiere a una sola posición de memoria. El tamaño de direcciones máximo de memoria tiene un límite el cual es

comúnmente definido por la palabra del chip. Los ejemplos de chip que aplican este esquema son Motorola 68K y Zilog Z80.

Típicamente las direcciones comienzan en cero y llegan hasta un valor máximo. A veces, particularmente en sistemas embebidos, la secuencia de direcciones puede ser discontinua. Mientras el programador entienda la arquitectura y tenga las herramientas de desarrollo correctas, esta discontinuidad no es problema.

2. ***Dividido en segmentos***

   Una desventaja percibida en la memoria plana es la limitación de tamaño, resuelto por el largo de una palabra. Un CPU de 16 bit podría solamente tener 64K de memoria y una arquitectura de 32 bit puede considerar más capacidad para muchas aplicaciones. La solución más común es utilizar memoria dividida en segmentos. Un ejemplo de memoria de esta arquitectura es el Intel 8086.

   La idea del registro de dirección de memoria es bastante simple. Las direcciones se dividen en dos partes: un número de segmento y una compensación. Las compensaciones (generalmente 16 bit) se utilizan la mayor parte del tiempo, donde los bit adicionales de categoría alta son sostenidos en uno o más registros especiales y asumidos para todas las operaciones. Para algunas direcciones de memoria en un rango grande, los registros del segmento deben ser recargados con un nuevo valor. Típicamente hay segmentos de registro individuales para código, datos y stack.

3. ***Cambio de Banco de direcciones***

   Otro acercamiento de extender el rango de direcciones de un CPU es el cambio de banco de direcciones (bank switching). Esta técnica es más compleja que la anterior, pero tiene la ventaja, de que cada memoria puede ser implementada con un procesador que no lo haga por sí mismo. Este esquema de memoria comprende dos partes: un rango de direcciones de memoria que presentan una “ventana” en una memoria más grande, y de un registro de control, que facilita el cambio de esta ventana. El acceso a la zona de memoria requiere que el registro de control sea verificado y ajustado si es necesario, antes de que la localidad requerida sea alcanzada dentro de la ventana.

4. ***Espacio Múltiple***

   Otro acercamiento más a incrementar el rango de direcciones de memoria de un número limitado, es el uso de las memorias múltiples.

   Por ejemplo, el Motorola 68000, opcionalmente puede tener cuatro espacios de direcciones: código de usuario y datos y supervisor de código y datos. Puesto que las funciones de estos espacios son esencialmente no-intercambiables, no se requiere ninguna disposición particular dentro de un lenguaje de alto nivel.

5. ***Virtual***

   El esquema de memoria virtual tiene una larga historia, pero, ha seguido apropiándose a los sistemas computacionales modernos. Donde la demanda de la memoria se extiende continuamente. Su empleo en sistemas embebidos, es realmente bajo, pero es importante conocer esta arquitectura. La memoria virtual da la impresión que la memoria tiene un tamaño indefinido. Esta impresión se da por el intercambio automático de zonas de la memoria por intervalos del disco duro usando partes especializadas de hardware. Un retardo ocurre cuando una locación de memoria que está actualmente fuera de intercambio debe ser leída. El retardo reduce la utilidad de la memoria virtual para sistemas de tiempo real.

2.3.16. La Influencia del Software en el Diseño de Hardware

¿Quién Diseña el Hardware?

El hardware se puede definir como la materia embebida, esta materia puede ser diseñada por una persona en una empresa pequeña o un pequeño departamento, esta definición va muy bien para el desarrollo de sistemas pequeños. Ya que esta persona puede entender muy bien los aspectos de uso y probablemente optimizar la interacción del soporte físico y por consiguiente el software, esto claro está, si se asume que esta persona es bastante experto en ambas disciplinas para evaluar las compensaciones correctamente.

En un laboratorio más grande, es mucho más probable hacer una diferenciación más aguda entre el soporte físico y el software, con tan solo una pequeña cantidad de expertos que conozcan ambos campos.

Las decisiones tomadas al principio de un proyecto de sistemas embebidos tienen implicaciones al final e incluso durante su producción. El **SKIMPING** en el hardware puede reducir el costo o energía, por ejemplo, podría costar tiempo extra programando al final. Sin embargo, el derrochar en el hardware, sobrestimando sus necesidades, carga a su diseño un costo adicional en cada unidad enviada. Los diseñadores tienen, generalmente, maestría en software o hardware, pero no en ambas.

2.3.17. Hardware llevando Software

Típicamente, el diseño de un sistema embebido comienza con el hardware. Solo cuando se ha diseñado el software entra en consideración. La eficiencia de la implementación sería realzada en última instancia si el software es considerado en un tiempo más temprano. Sería ridículo sugerir que los ingenieros de software hicieran el trabajo de los ingenieros de software, pero, su implicación temprana sería útil.
Sin nada más, reduciría al mínimo el “dedo que señala” durante la fase de integración, cuando cada “lado” culpa al otro por cualquier problema que se presente.

2.3.18. Compensaciones Software/ Hardware

La consideración de los requisitos y las capacidades del software tiene una verdadera influencia en muchos puntos durante la definición del hardware.

2.3.19. Selección del Microprocesador

Un factor en la selección del microprocesador es la disponibilidad de las herramientas de software avanzado para el desarrollo. Aunque los chips de baja potencia están especificados para muchos usos, el ahorro de unos cuántos microwatts del consumo de energía de un CPU, es el uso de memoria adicional necesario para almacenar código ineficiente.

2.3.20. Tamaño y Mezcla de Memoria

Una decisión sobre la cantidad exacta de memoria y la mezcla de ROM (flash, etc) y de RAM debe ser hecha y atrasada como sea posible en el ciclo de diseño, puesto que esto es a menudo difícil de predecir. El código y los datos que abarrotan dentro de un área de memoria demasiado pequeña, es un problema, y el tener mucha memoria implica en el precio y energía. A veces, los sitios de la memoria pueden ser diseñados para tomar cualquier RAM o ROM, lo cual significa permitir los trazos de la señal de escritura. Esta estrategia permite que el tipo de mezcla de memoria se determine más tarde en el ciclo de diseño y ofrece la habilidad de reemplazar más de la ROM con RAM, para propósitos de depuración.

2.3.21. Implementación de Periféricos

La inclusión de periféricos en el diseño debe ser considerada cuidadosamente. Por ejemplo, los contadores de tiempo y la entrada/ salida serial, pueden ejecutarse en software cuando son necesarios. Hay una mayor carga en software cuando cada bit de la comunicación serial en lugar de tener manejo en hardware del byte. Sin embargo, si la capacidad de cálculo de CPU está disponible y el costo del hardware es una preocupación, teniendo el software que haga el trabajo es una solución razonable. Después de todo uno paga el desarrollo del programa solo una vez, pero, un chip adicional agrega costo a cada unidad enviada. Esta compensación se debe hacer cuidadosamente, porque las

consideraciones de un uso de poco volumen (donde el costo del desarrollo del software es dominante) difieren de esas para el envío de un producto en gran volumen (donde el costo por unidad puede disminuir).

2.3.22. Depuración de Hardware

La adición de hardware, para ayudar a eliminar errores, esto no se les ocurre a muchos diseñadores de hardware. Es una parte de digna importancia de considerar a futuro.

2.3.23. Emuladores en Circuito (ICEs)

Los ICEs históricamente eran un instrumento optativo para el desarrollo de software embebido. Ellos proporcionaban una manera totalmente inviable de eliminar errores de código a alta velocidad en un objetivo real. Pero, como la complejidad del procesador aumentó, y más importante, las frecuencias de reloj se han incrementado, este método se volvió más y más costoso y su disponibilidad disminuyó. Ahora sólo es factible realizarlo solamente para dispositivos pequeños. Si un ICE está disponible y el costo no es muy alto, este puede ofrecer una herramienta puesta a punto mucho mejor. Sin embargo, en realidad un equipo completo de desarrolladores de software raramente puede ser equipados con ICEs para la depuración del software. Estos instrumentos son ideales para direccionar problemas complejos resultantes de la interacción cerrada del software y el hardware, y si son de suministros cortos, debe ser reservado para este tipo de trabajo.

2.3.24. Monitoreo de Errores

Una buena solución, en muchos casos, es hacer uso del monitoreo de errores. Esto requiere de una disposición en el hardware objetivo: ROM debe ser reemplazado temporalmente por RAM y un puerto serial adicional u otro dispositivo de entrada/salida debe estar disponible en el tiempo de depuración.

Ni uno ni otro de estos requisitos presentan un problema en más circunstancias si la necesidad es considerada tempranamente en la fase del diseño. El resultado es la habilidad para depurar el código a una alta velocidad. Esto no tiene todas las facilidades de un ICE, pero si para la mayoría de los propósitos. En particular, una depuración basada en monitoreo puede ser la base de un modo de implementación sin errores con facilidad en un sistema operativo en tiempo real.

2.3.25. Soporte de Depuración en Chip

Muchos de los microprocesadores modernos proveen soporte en chip para la depuración del software, generalmente, llamado "*depuración en chip*" (OCD). La familia del microcontrolador Freescale 683xxx (basados en el núcleo del CPU32; excepto el 68302), por ejemplo, tiene modo de depuración de fondo (BDM). Muchos otros dispositivos utilizan una interface JTAG, con un resultado similar.

OCD generalmente es implementado en el microcódigo del chip ofrece una capacidad de depuración anti intruso. Para usar el OCD, un adaptador es necesario entre el adaptador y la máquina para manejar el protocolo especial de sincronización y administrar la depuración en una interface de un lenguaje de alto nivel.

Conectando este adaptador a la tarjeta objetivo requiere un conector apropiado, pero el costo y el espacio para agregarlo son insignificantes. Sin embargo esto debe ser considerado durante la fase de diseño.

2.3.26. Soporte de Autoprueba

La mayoría de los sistemas embebidos tienen un grado de autoprueba incorporada. Algunas veces la autoprueba es ejecutada solo cuando al chip se le suministra la energía requerida para su funcionamiento, y llevar acabo el chequeo de fallas que pueda tener. En

otros sistemas, el código de antiprueba puede funcionar como una tarea de fondo cuando no es requerido algún otro proceso.

2.3.27. Circuitos de Entrada/Salida

La autoprueba facilita que se pueda incorporar dispositivos específicos de entrada/salida. Un ejemplo típico es una interfaz serial (RS232). Para confirmar que el chip con la interfaz en serie recibe y transmite correctamente, se puede ejecutar un "*loopback*". Esta capacidad da lugar a cada carácter transmitido que es enviado directamente al receptor. Primero, el código de autoprueba envía una secuencia y enseguida verifica que está recibiendo correctamente. El canal del loopback en el circuito de la entrada/salida debe ser controlable por el software.

2.3.28. Interruptores a Bordo

Un interruptor a bordo, los puentes, o los botones pueden proporcionar comandos de prueba al software. Con un banco de cuatro interruptores. Por ejemplo, un interruptor podría activar el modo de autoprueba y los otros tres podrían seleccionar uno de ocho posibles frecuencias para una línea serial.

2.3.29. Estado de Displays

Un display puede ser tan complejo como una línea de caracteres puede escribir un mensaje, o tan simple como un led, el cual puede ser encendido o apagado por el software. Una cantidad sorprendente de información puede ser trasportada por tan solo un led. Puede ser en uno de tres estados simples: encendido, apagado o destellando. Una buena opción es usar el destello para indicar que el software está en buen estado, porque este requiere mantener los destellos y, por lo tanto, a prueba de averías. Por supuesto, las instrucciones de encendido/apagado del led idealmente deben estar embebidas en la parte principal del

software. Los estados fijos encendido/apagado son usados para indicar posibles condiciones de fallo. Como los focos de casa, la manera de los destellos del led puede indicar un estado. Hay dos variaciones de destellos que pueden ser empleadas: los destellos son separados por pausas (donde el número de destellos puede indicar un estado particular) o los tiempos variables de utilización (periodos) pueden transportar diversos estados. Cada uno de estos tiene sus atracciones, pero los ciclos de tiempo de utilización es probablemente un poco más fácil de ejecutar y de entender.

2.4. HARDWARE EMBEBIDO [15]

Esta sección es de suma importancia tanto para el ingeniero de hardware embebido como para el ingeniero de software embebido. Es importante que todo diseñador de sistemas embebidos entienda los diagramas y símbolos que los ingenieros de hardware utilizan para describir sus diseños al mundo exterior. Estos diagramas y símbolos son clave para el entendimiento de forma rápida y eficiente hasta un diseño con complejidad alta, sin importar cuanta experiencia práctica se tenga en el diseño de hardware. También contienen la información que un programador embebido necesita para diseñar cualquier software que requiera compatibilidad con el hardware y le proporcionan el cómo realizar una comunicación exitosa del hardware al software. Hay distintos tipos de diagramas en ingeniería de hardware:

- ***Diagramas a bloques***: Típicamente representan los componentes principales de una tarjeta (procesadores, bus de datos, entradas y salidas, memoria) o de un solo componente (por ejemplo, un procesador), en una arquitectura de un sistema de alto nivel.

- ***Diagramas Esquemáticos***: Son diagramas de circuitos electrónicos que proveen una vista más detallada de todos los dispositivos dentro de un circuito o dentro de un solo componente, todo desde procesadores hasta resistencias.

- ***Diagramas Eléctricos***: Estos diagramas representan las conexiones del bus entre los componentes principales y de menor importancia en un tablero o dentro de un chip. En éstos diagramas, las líneas verticales y horizontales se utilizan para representar las líneas de un bus, y se utilizan los símbolos esquemáticos. Pueden representar una pintura aproximada de la disposición física de un componente o una tarjeta.

- ***Diagramas Lógicos***: Se utilizan para demostrar una gran variedad de información del circuito usando símbolos lógicos (AND, OR, NOT, XOR) y las entradas/salidas lógicas (1 y 0). Estos diagramas no substituyen diagramas esquemáticos, sino que pueden ser útiles en la simplificación de cierto tipo de circuitos para entender cómo funcionan.

- ***Cronogramas***: Exhiben los gráficos de la sincronización de las distintas variables de entrada y salida de un circuito, así como las relaciones entre ellas. Son los diagramas más comunes de manuales de usuario y hojas de datos del hardware.

2.4.1. La Tarjeta Embebida y el Modelo Embebido

En dispositivos, todo el hardware electrónico reside en una tarjeta, llamada *PW* por sus siglas en inglés "*Printed Wiring Board*" (tarjeta de cableado impreso), o *PCB* "*Printed Circuit Board*" (tarjeta de circuito impreso). Los PCB, los cuáles, a menudo son hechos con placas finas de fibra de vidrio. La trayectoria eléctrica del circuito se imprime en cobre, el cual lleva las señales eléctricas entre los distintos componentes conectados en la tarjeta. Todos los componentes electrónicos que componen que componen el circuito van conectados a la tarjeta, ya sea soldado o colocado en una base; comúnmente llamado "*zócalo*", u otro mecanismo de la conexión. Todo hardware en una tarjeta embebida, se encuentra localizado en la capa de hardware en el modelo de sistemas embebidos, el cual se muestra en la *figura 2.3*.

Figura 2.3. Modelo de Sistemas Embebidos y la Tarjeta Embebida

En el nivel más alto, el mayor componente de hardware de más tarjetas puede ser clasificado en cinco categorías mayores:

1. *Unidad Central de Procesamiento* (CPU). Procesador maestro.
2. *Memoria.* Donde el sistema de software es almacenado.

3. *Dispositivos de Entrada*. Entrada de procesadores esclavos y componentes eléctricos relativos.

4. *Dispositivos de Salida*. Salida de procesadores esclavos y componentes eléctricos relativos.

5. *Camino de datos o Buses*. Interconecta los otros componentes, proporcionando un "*camino*" para el viaje de los datos de un componente a otro, incluyendo cualquier cable, puentes de bus y/o controladores de bus.

Estas cinco categorías están basadas en los principales elementos definidos por el *Modelo de Von Neumann*, ver *figura 2.4*, una herramienta que puede ser usada para entender cualquier arquitectura de hardware del dispositivo electrónico. El modelo de Von Neumann es un resultado del trabajo publicado por John Von Neumann en 1945, el cual, definió los requerimientos de una computadora electrónica de propósito general. Porque

sistemas embebidos, son un tipo de sistema computacional, este modelo puede ser aplicado como medio para la comprensión del hardware en sistemas embebidos.

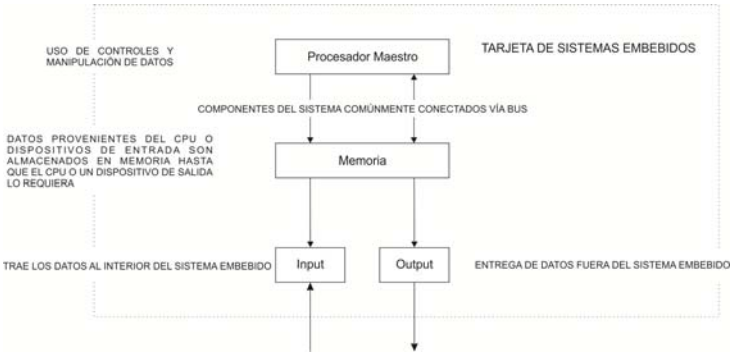


Figura 2.4. Organización de la Tarjeta Embebida

En la *figura 2.5* se muestra un ejemplo de un diagrama de bloques de una tarjeta de desarrollo con FPGA. La cual es una descripción general de los componentes que hacen posible implementar varios proyectos de control, inteligencia artificial, y en el caso de este proyecto procesamiento de imágenes.

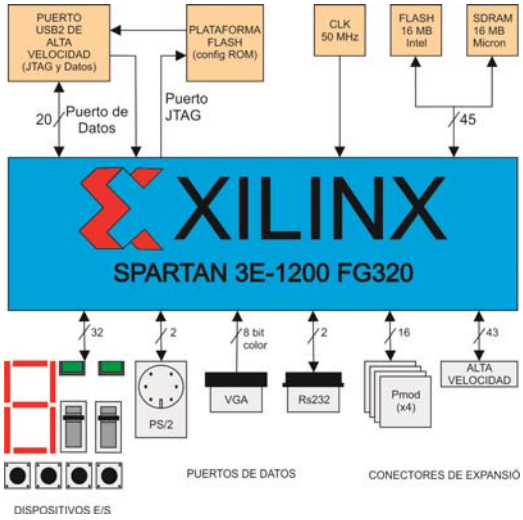


Figura 2.5. Diagrama de Bloques de la Tarjeta de Desarrollo

2.4.2. Arreglo de Compuerta Programables en Campo (FPGA)

Un FPGA, es un semiconductor en el cual el circuito de lógica actual puede ser modificado a las necesidades del desarrollador, el chip es relativamente barato. Es

importante observar la diferencia entre la programación y la programación lógica, o diseño lógico, como generalmente se le llama: un programa de software siempre necesita funcionar en algún microcontrolador con una apropiada arquitectura de sistema de instrucción (ISA), mientras que un programa lógico es el microcontrolador. De hecho este programa lógico puede especificar un controlador que acepte como entrada un ISA en particular. Esto es llamado "*soft core*", construido de bloques generales, estos mejor que usar el derecho de propiedad intelectual, se pueden comprar de compañías como Xilinx, Inc., Altera Corporation, Cypress, etc. Entonces se transfieren al FPGA donde ejecutan la funcionalidad deseada. Algunos FPGA modernos integran plataformas o procesadores duros multiuso en la lógica como lo es una arquitectura PowerPC o DSP. Otros módulos comunes, duros y suaves, incluye los multiplicadores, la interfaz lógica, y los bloques de memoria.

El *diseño lógico* determina la funcionalidad del FPGA. Esta configuración es escrita y retenida hasta que es borrada. Hay tres tipos de FPGA que son:

- ***Antifuse***. No son programables

- ***SRAM***. El programa es volátil; debe ser programado encendido.

- ***Flash***. Es no volátil, esto significa que el diseño lógico permanece en el chip a través de ciclos de energía.

Una enorme ventaja del FPGA es la gran flexibilidad en la lógica, ofreciendo extremo paralelismo en el flujo y el proceso de los datos en el uso de visión. Por ejemplo, se puede crear 320 acumuladores y ALUs en paralelo, suministrando una imagen entera de 320x240 en 240 ciclos de reloj. Otro ejemplo, sería poner una región de interés en el FPGA y después realizar operaciones en los pixeles de la región entera simultáneamente. El FPGA puede alcanzar velocidades cercanas al DSP y ASIC. Requiere un bit de más energía que un ASIC. Tiene bajos costos de ingeniería no-recurrente. Pero precios en volumen más altos.

Usualmente los desarrolladores de algoritmos e ingenieros de software son entrenados en un modelo secuencial. Eso y la flexibilidad de la circuitería lógica, hace que el diseño paralelo sea un reto, particularmente porque la mejor implementación no es evidentemente intuitiva. Una de las primeras dificultades entre el FPGA y el GPP, y entre el FPGA y el CPU, y entre otras plataformas posibles. Comúnmente "los lenguajes descriptivos de hardware" son Verilog y VHDL.

El FPGA es un gran recurso para el paralelismo y ofrece tremenda flexibilidad en sistemas de visión embebida. Por otro lado, el FPGA es absolutamente hambriento de energía y su frecuencia de reloj más baja que la de un DSP. Hoy en día hay disponible una amplia variedad de FPGA en el mercado comercial. La elección óptima para un sistema de visión embebida es una combinación de un simple FPGA con suficientes entradas/salidas de propósito general como recursos para manejar las imágenes de las interfaces entrantes y salientes más una interface de 64 bit al DSP. El criterio de selección igualmente importante será la cantidad de memoria embebida en el FPGA así como el encapsulado. La mayoría de FPGAs tienen una abundante capacidad de entrada/salida con respecto a lógica interna así que es probable que una interfaz opcional de 32 bits SDRAM, puede ser posible. Tal interfaz en un sistema embebido de visión proveería al FPGA de acceso privado a su propia área de almacenamiento en el costo de tiempo de acceso y añadido a la complejidad del FPGA. Los recursos abundantes de entradas/salidas también son usados para dejar el control de las compuertas de entrada a otro chin en la tarjeta.

2.5. VISIÓN EMBEBIDA POR COMPUTADORA [16]

Al verse con la tarea de desarrollar un sistema embebido que realice múltiples tareas de visión por computadora. Uno de los puntos más importantes tal vez sería pensar en los componentes de hardware a utilizar. Hay una gran variedad de arquitecturas disponibles, tales como Procesadores de Señales Digitales (DSPs), Arreglos de Compuertas Programables en Campo (FPGA), Sitemas en un Chip (SoCs), Circuitos Integrados de Aplicación Específica (ASICs), Procesadores de Propósito General (GPPs), Unidades de Procesamiento Gráfico (GPUs), sin mencionar los bloques que interconectan y de memoria.

Para apreciar completamente los desafíos y potenciales para los sistemas digitales de visión, se presenta un *proceso paralelo* en tiempo real, desde el fotón al fotodiodo, desde la carga del acumulador a la lectura de la circuitería de salida, desde el chip sensor sobre el cableado a la tarjeta del chip de procesamiento, detallando el flujo de datos y el almacenamiento de datos en la tarjeta, y cómo los parámetros del sensor son controlados.

Solamente comprendiendo este proceso y los pasos involucrados en cada etapa puede ser optimizado completamente, insertar el procesamiento y almacenamiento en lugares óptimos, los datos que incurren antes innecesariamente en el costo computacional. La meta es alcanzar el mejor la mejor compensación entre varias metas en conflicto, las variables dependientes con las cuales son tratadas típicamente en sistemas embebidos son:

- Funcionamiento de la aplicación
- Velocidad
- Disipación de energía
- Tamaño del sistema

Se cree que tal vista completa e integrada de hardware y software de visión por computadora, a menudo se pasa por alto y por lo tanto su potencial no ha sido alcanzado completamente debido a las distintas comunidades de ingenieros electrónicos e ingenieros computacionales.

Ya que un sistema embebido comprende uno o múltiples procesadores que sirven a una tarea específica en un sistema complejo. Por ejemplo: procesadores de video en televisiones y reproductores de DVD. Sus ventajas caen en la conveniencia de la tarea, resultando de gran velocidad, bajo consumo de energía, costo reducido y arranque más rápido que los sistemas de propósito general. Los sistemas embebidos, particularmente, son adaptados a las corrientes de datos a alta velocidad en programas pequeños. Los buses

dedicados son usados para evitar competir con otras necesidades del proceso. Los aceleradores del hardware pueden procesar datos en una manera altamente paralela, con velocidades sorprendentes, y sin interferir con las tareas de otro procesador.

Los factores de forma de las computadoras embebidas son que nunca encogen, con más y más componentes integrados en el mismo chip. Un indicador fuerte para la importancia cada vez mayor de los sistemas para el análisis de datos en tiempo real es: la integración de las capacidades multimedia tales como convertidor analógico/digital, la integración de las capacidades del tratamiento de señales, y la puesta en práctica de los sistemas de instrucción que prevén una necesidad de tal proceso. Con la ayuda de éstos altamente, las capacidades de proceso optimizadas, los usos de la visión por computadora están encontrando camino dentro de los productos de consumo.

CAPÍTULO III
CONTROLADOR DE SÍNCRONÍA VGA

3.1. INTRODUCCIÓN [17]

VGA es un estándar de video introducido en los 80's por IBM PCs, muy utilizado en la actualidad en las tarjetas gráficas para PCs y monitores, se manejan 8 colores básicos para una resolución de 640x480 pixeles, utilizando monitores CRT (Cathode Ray Tube). El cual, es utilizado para las pruebas de este proyecto. Es necesario conocer el modo en que se realiza el barrido horizontal y vertical para la exhibición de una imagen en el monitor. Este es el campo de estudio descrito en este capítulo, lo cual, es una parte esencial del proyecto a desarrollar en este trabajo de tesis.

3.2. OPERACIÓN BÁSICA DE UN MONITOR

El diagrama conceptual de un monitor monocromático se muestra en la *figura 3.1*. El cañón de electrones (cátodo) genera un haz enfocado, el cual atraviesa un tubo de vacío y golpea eventualmente la pantalla fosforescente, la luz es emitida al instante que los electrones golpean un punto de fósforo en la pantalla. La intensidad del haz y el brillo del punto, son determinados por el nivel de voltaje de la señal externa de la entrada de video, etiquetado como **mono** en la *figura 3.1*. La señal mono, es una señal analógica cuyo nivel de voltaje se encuentra entre 0 y 0.7 V.

Dos bobinas de desviación (una horizontal y la otra vertical) fuera del tubo producen campos magnéticos, para controlar cómo viaja el haz y determinar el punto de la pantalla donde golpeará. En monitores actuales, el haz atraviesa la pantalla sistemáticamente en un patrón fijo, el cual es, de izquierda a derecha y de arriba abajo, como se muestra en la *figura 3.2*.

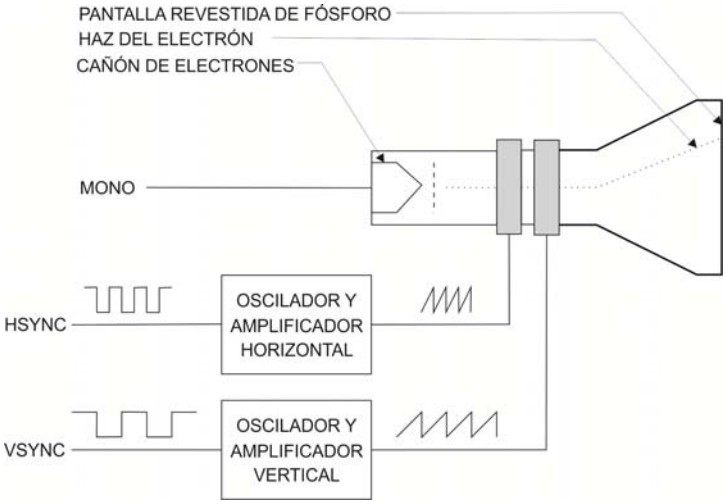


Figura 3.1 Diagrama Conceptual de un Monitor CRT

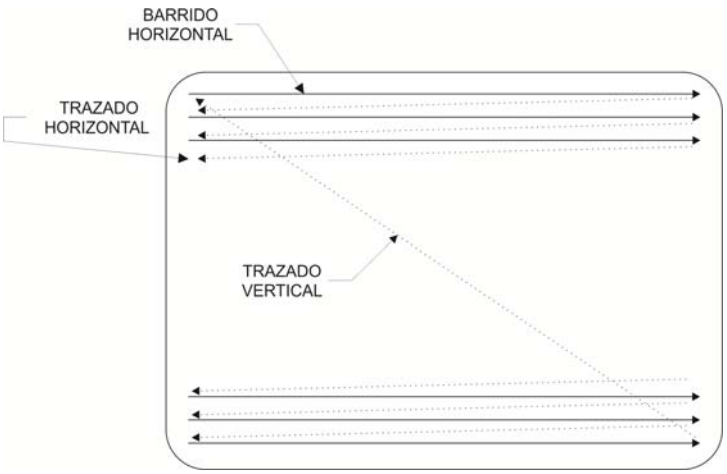


Figura 3.2 Patrón de Barrido CRT

Los osciladores y amplificadores internos de un monitor generan formas de onda de diente de sierra para controlar las dos bobinas. Por ejemplo, el haz se mueve desde el borde izquierdo al borde derecho, como el voltaje aplicado a la bobina de desviación horizontal aumenta gradualmente. Después de alcanzado el borde derecho, el haz vuelve rápidamente al borde izquierdo (es decir, retrazo) cuando el voltaje cambia a 0. La relación entre la forma de onda de diente de sierra y el barrido se muestran en la *figura 3.4*. Dos señales externas de sincronización, HSYNC y VSYNC, controlan la generación de la forma de onda de diente de sierra. Estas señales son digitales. La relación entre la señal HSYNC y el diente de sierra horizontal también se muestra en la *figura 3.4*. Notando que los periodos 1

y 0 de la señal HSYNC corresponden a las rampas de levantamiento y que caen de la forma de onda de diente de sierra.

La operación básica de un monitor CRT a color es similar, con excepción de que tiene tres haz de electrones, que se proyectan a los puntos rojos (R), verdes (G) y azules (B) de los puntos de fósforo de la pantalla, los tres puntos se combinan para formar un pixel. Podemos ajustar los niveles de voltaje de las tres señales de entrada de video para obtener el color deseado en el pixel.

3.3. PUERTO VGA

El puerto VGA es la interface entre el monitor y la tarjeta de desarrollo de sincronía, este puerto puede tener varias configuraciones dependiendo la resolución de color que se requiera algunas de estas configuraciones se ven a lo largo de esta sección. En la actualidad, la configuración de un puerto VGA en un tarjeta de video de una PC es de 32 bit, lo cual, permite tener una combinación de intensidades de cada color de gran tamaño. A continuación se habla de la resolución de color RGB de tres tarjetas de desarrollo.

3.3.1. Puerto VGA de la tarjeta Spartan 3 de Xilinx

El puerto VGA tiene cinco señales activas, incluyendo la sincronía horizontal y vertical (HSYNC, VSYNC), y tres señales de video: haz rojo, verde y azul, físicamente dirigido al conector de video tipo D de 15 pines. Una señal de video es analógica y el controlador de video utiliza un convertidor digital a analógico, para convertir la salida digital al nivel analógico deseado. Las tres señales de video (RGB) pueden generar 2^3 colores diferentes. En la tarjeta, un bit es usado por cada señal de video, esto nos da hasta un total de 8 colores posibles. La combinación de los 8 colores se muestra a continuación en la *tabla 3.2*.

Tabla 3.1. Combinación de colores VGA de tres bit

ROJO(R)	VERDE(G)	AZUL(B)	COLOR RESULTANTE
0	0	0	NEGRO
0	0	1	AZUL
0	1	0	VERDE
0	1	1	CYAN
1	0	0	ROJO
1	0	1	MAGENTA
1	1	0	AMARILLO
1	1	1	BLANCO

Si usamos la misma señal de 1 bit para manejar las señales de video, se convierten en cualquier “000” o “111” y se mostraría blanco y negro como monitor monocromático.

3.3.1. Puerto VGA de la tarjeta NEXYS 2 de Dgilent

A diferencia de la tarjeta Spartan 3, la tarjeta Nexys 2, en su puerto VGA tiene 10 señales activas, incluyendo la sincronía horizontal y vertical (HSYNC, VSYNC), y tres señales de video: haz rojo, verde y azul. La diferencia es el tamaño de las señales RGB, ya que para esta tarjeta se tiene 3 bit para color rojo, 3 bits para color verde y 2 bits para color verde, pero físicamente trabaja igual que el puerto VGA de la Spartan 3. Las tres señales de video (RGB) pueden generar 2^8 colores diferentes. En la tarjeta, 3 bits son usados por las señales R y G, la señal B usa 2 bits de video, esto nos da hasta un total de 256 colores posibles y nos permite cambiar el tono de cada color dependiendo de la combinación que se use. La combinación básica de colores se muestra a continuación en la *tabla 3.3*.

Tabla 3.2. Combinación de colores VGA de 8 bits

ROJO(R)	VERDE(G)	AZUL(B)	COLOR RESULTANTE
000	000	00	NEGRO
000	000	11	AZUL
000	111	00	VERDE
000	111	11	CYAN
111	000	00	ROJO
111	000	11	MAGENTA
111	111	00	AMARILLO
111	111	11	BLANCO

3.3.2. Puerto VGA de la tarjeta SPARTAN 3AN de Xilinx

A diferencia de las dos tarjetas anteriores, la tarjeta Spartan 3AN, en su puerto VGA tiene 14 señales activas, incluyendo la sincronía horizontal y vertical (HSYNC, VSYNC), y tres señales de video: haz rojo, verde y azul. La diferencia es el tamaño de las señales RGB, ya que para esta tarjeta se tiene 4 bit para cada señal, físicamente trabaja igual que los puertos VGA de la tarjeta Spartan 3 y la tarjeta Nexys 2. Las tres señales de video (RGB) pueden generar 2^{12} colores diferentes, esto nos da hasta un total de 4096 colores posibles y nos permite cambiar el tono de cada color dependiendo de la combinación que se use. La combinación básica de colores se muestra a continuación en la *tabla 3.4*.

Tabla 3.3. Combinación de colores VGA de 12 bits

ROJO(R)	VERDE(G)	AZUL(B)	COLOR RESULTANTE
0000	0000	0000	NEGRO
0000	0000	1111	AZUL
0000	1111	0000	VERDE
0000	1111	1111	CYAN
1111	0000	0000	ROJO
1111	0000	1111	MAGENTA
1111	1111	0000	AMARILLO
1111	1111	1111	BLANCO

3.4. CONTROLADOR DE VIDEO

Un controlador de video genera las señales de sincronización de los pixeles y los datos de salida.

En la *figura 3.3* se muestra un diagrama simplificado de un controlador VGA. Contiene un circuito de sincronización, etiquetado VGA-SYNC, y un circuito de la generación de pixel.

El circuito VGA-SYNC, genera el tiempo y las señales de sincronización, las señales HSYNC y VSYNC están conectadas al puerto VGA para controlar el barrido horizontal y vertical del monitor. Las dos señales son decodificadas desde los contadores internos las cuales son **PIXEL_X** y **PIXEL_Y,** que indican la posición del barrido y esencialmente la

localización del pixel actual. El circuito también genera la señal **VIDEO_ON,** que indica cuando habilitar o deshabilitar el display.

El circuito que genera el pixel por medio de las tres señales de video, a las cuales se les refiere en conjunto como señal **RGB**. Un valor de color se obtiene según las coordenadas actuales del pixel (PIXEL_X, PIXEL_Y), el control externo y las señales de datos.

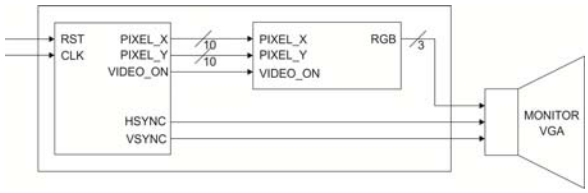


Figura 3.3 Diagrama de Bloques de un Controlador VGA

3.5. SINCRONIZACIÓN VGA

El circuito de sincronización de video genera la señal HSYNC, la cual específica el tiempo requerido para el barrido de una fila, y la señal VSYNC, que específica el tiempo requerido para el barrido de la pantalla completa. Basándose en una resolución VGA de 640x480 con una frecuencia de 25MHz, es decir, 25 millones de pixeles son procesados en un segundo.

La pantalla de un monitor CRT usualmente incluye un pequeño borde negro, como se muestra en la figura 4. El rectángulo medio es la porción visible. Se observa que la coordenada del eje vertical aumenta hacia abajo, las coordenadas de las esquinas superior-izquierda e inferior-derecha son (0.0) y (639, 479), respectivamente.

3.5.1. Sincronización Horizontal

En la *figura 3.4* se muestra un diagrama detallado del tiempo de un barrido horizontal un periodo de la señal **HSYNC** contiene 800 pixeles y pueden ser divididos entre cuatro regiones:

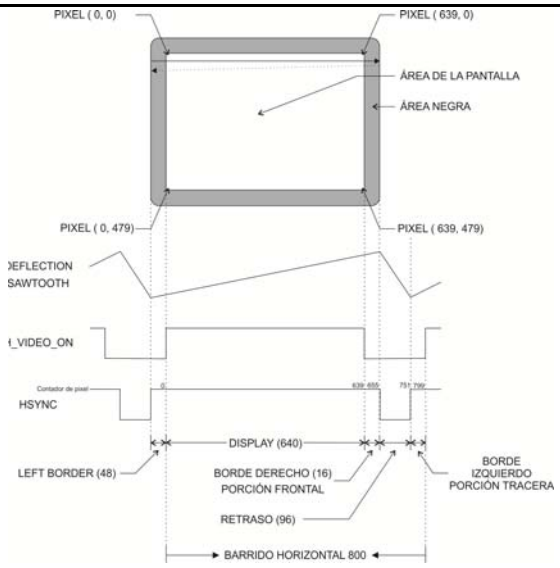


Figura 3.4 Diagrama de Barrido Horizontal

- ***Display o Exhibición:*** región donde se muestran los pixeles en la pantalla. La longitud de esta región es de 640 pixeles.

- ***Retrazo o Trazo:*** región en la cual el haz del electrón regresa a al extremo izquierdo. La señal de video debe estar deshabilitada y la longitud de esta región es de 96 pixeles.

- ***Borde Derecho:*** región que forma el borde derecho de la región de la pantalla, también conocido como la *porción frontal* (porción antes del retrazo). La señal de video debe estar deshabilitada, y la longitud de esta región es de 16 pixeles.

- ***Borde Izquierdo:*** región que forma el borde izquierdo de la región de la pantalla. También conocida como *porción trasera* (porción después del retrazo). La señal de video debe estar deshabilitada, y la longitud de esta región es de 48 pixeles.

Observar que la longitud del borde derecho e izquierdo puede variar para los diferentes tipos de monitores.

La señal **HSYNC** puede obtenerse por un contador especial mod800 y un circuito decodificador.

Las cuentas son hechas en la parte superior de la señal **HSYNC** en la *figura 3.4*. Se trabajará iniciando la cuenta al inicio de la región de la pantalla. Esto permite usar la salida del contador como la coordenada horizontal (eje X), esta salida constituye la señal **PIXEL_X**. La señal **HSYNC** va en estado bajo, cuando la salida del contador se encuentra entre 656 y 751.

Observe que el monitor CRT debe ser negro en los bordes izquierdo y derecho.
Se usa la señal **h_video_on** para indicar si la coordenada actual está en la región mostrable, se afirma solamente cuando la cuenta del pixel es más pequeña de 640.

3.5.2. Sincronización Vertical

El haz del electrón se mueve gradualmente de arriba abajo y regresa hacia arriba. Esto corresponde al tiempo requerido para refrescar la pantalla completa el formato de la señal **VSYNC** es similar al de la señal **HSYNC**, como se muestra en la *figura 3.5*.

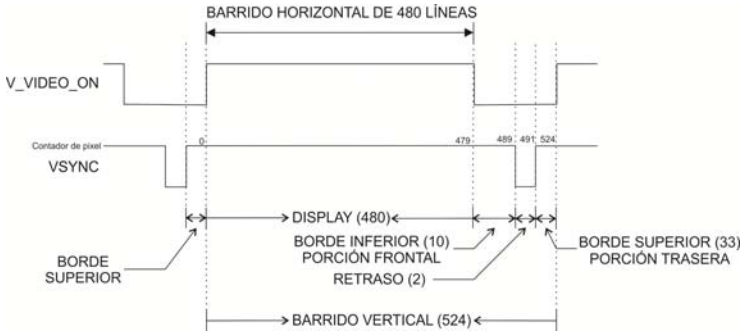


Figura 3.5 Diagrama de Tiempos de un Barrido Vertical

La unidad de tiempo del movimiento es representado en términos del barrido horizontal. Un periodo de la señal **VSYNC** es de 525 líneas y pueden ser divididas en cuatro regiones:

- ***Display o Exhibición:*** es la región donde las líneas horizontales actuales son exhibidas en pantalla, la longitud de esta región es de 480 líneas.

- ***Retrazo o Trazo:*** región en la que el haz del electrón regresa a la parte superior de la pantalla. La señal de video debe estar deshabilitada, y la longitud de esta región es de 2 líneas.

- ***Borde Inferior:*** región que forma la parte inferior del área exhibida. También es conocida como porción frontal. La señal de video debe estar deshabilitada, y la longitud de esta región es de 10 líneas.

- ***Borde Superior:*** región que forma la parte superior del área exhibida. También conocida como la porción de vuelta (después del retrazo). La señal de video debe estar deshabilitada, y la longitud es de 33 líneas.

Como en el barrido horizontal, las longitudes de las partes superior e inferior, puede variar para los diferentes monitores.

La señal **VSYNC** puede ser obtenida por un contador especial mod525 y un circuito decodificador. Nuevamente se iniciará contando desde el inicio del área exhibida, esto permite usar la salida del contador como la coordenada vertical (eje Y).
Esta salida constituye la señal **PIXEL_Y**, la señal **VSYNC** se encuentra en estado bajo cuando la línea del contador se encuentra entre 490 o 491. Como en el barrido horizontal, se usa la señal **VIDEO_ON** para indicar cuál es la coordenada para indicar si la coordenada vertical se encuentra dentro del área exhibida, se afirma solamente cuando la cuenta de las líneas es menor que 480.

3.5.3. Cálculo de Tiempo de las Señales de Sincronización VGA

Como fue mencionado, se asume que la velocidad del pixel es de 25 MHz, es determinado por tres parámetros:

- *p:* es el número de pixeles en una línea horizontal. Para la resolución 640x480, es:

$$p = 800 \frac{pixeles}{línea} \qquad (3.1)$$

- *l:* número de líneas en una pantalla (barrido vertical). Para la resolución 640x480, es:

$$l = 525 \frac{líneas}{pantalla} \qquad (3.2)$$

- *s:* número de pantallas por segundo para la operación oscilar-libre, podemos fijarlo de la siguiente manera:

$$s = 60 \frac{pantallas}{segundo} \qquad (3.3)$$

El parámetro ***s*** especifica la rapidez en la que la pantalla debe ser actualizada. Para el ojo humano, la rapidez de actualización debe ser por lo menos 30 pantallas por segundo para aparentar que el movimiento es continuo. Para reducir el oscilador, el monitor tiene una velocidad más alta, tal como 60 pantallas por segundo. La velocidad del pixel se puede calcular por los tres parámetros:

$$pixel\ rate = p * l * s \approx 25M \frac{pixeles}{segundo} \qquad (3.4)$$

La velocidad del pixel y la velocidad de actualización pueden calcularse de manera similar. Claramente, la velocidad incrementa como la resolución y la actualización.

3.6. DIAGRAMAS A BLOQUES DEL CONTROLADOR VGA EN FPGA

El circuito de sincronización de video genera la señal HSYNC, la cual específica el tiempo requerido para el barrido de una fila, y la señal VSYNC, que específica el tiempo requerido para el barrido de la pantalla completa. Basándose en una resolución VGA de 640x480 con una frecuencia de 25MHz, es decir, 25 millones de pixeles son procesados en un segundo.

La pantalla de un monitor CRT usualmente incluye un pequeño borde negro, como se muestra en la *figura 3.4*. El rectángulo medio es la porción visible. Se observa que la coordenada del eje vertical aumenta hacia abajo, las coordenadas de las esquinas superior-izquierda e inferior-derecha son (0.0) y (639, 479), respectivamente [Pong, 2008].

A continuación se representan los diagramas a bloques de la sincronización VGA para n bit de color; esto debido a que nuestra descripción puede ser fácilmente adaptada a cualquier resolución de color. Los diagramas son diseñados de acuerdo a la metodología TOP-DOWN [Romero, 2007], comenzando por el diagrama de más alto nivel, el cual se muestra en la figura 3.6. Donde se observa que está formado por dos componentes: VGASYNC y Generador de Imagen RGB. El circuito VGASYNC se compone de varios circuitos para lograr la sincronía y el segundo, únicamente es un módulo que utiliza la sincronía para lograr mostrar un color en todo el monitor.

La función del circuito VGASYNC fue presentada en la sección anterior. Si la frecuencia de reloj del sistema es de 25 MHz, el circuito puede ser implementado por dos contadores especiales:

- Un contador mod800 para el barrido horizontal
- Un contador mod525 para el barrido vertical.

Puesto que los diseños utilizados generalmente en la tarjeta utilizan 50 MHz, la frecuencia del reloj del sistema es dos veces la frecuencia del pixel. En lugar de crear un dominio separado del reloj de 25 MHz, y por lo tanto, no respetar la metodología de diseño síncrona, se puede generar una señal de habilitación o pausa del conteo a 25MHz. La señal es llevada al puerto PT como una señal de salida a la operación coordenada del circuito generador del pixel.

El código HDL es mostrado en el ***listado 3.1***. Consiste de un contador **mod2**, el cual genera la habilitación a 25MHz y dos contadores para el barrido horizontal y vertical.
Los valores de varias regiones del barrido horizontal y vertical son definidos como constantes. Pueden ser fácilmente modificados si se usa una frecuencia y resolución diferentes. Para eliminar interferencias potenciales, se insertan **buffers** de salida para las señales **HSYNC** y **VSYNC**. Esto lleva a un retardo de un ciclo de reloj. Se debe agregar un buffer similar para la señal **RGB** en el circuito generador de pixel para compensar el retardo.

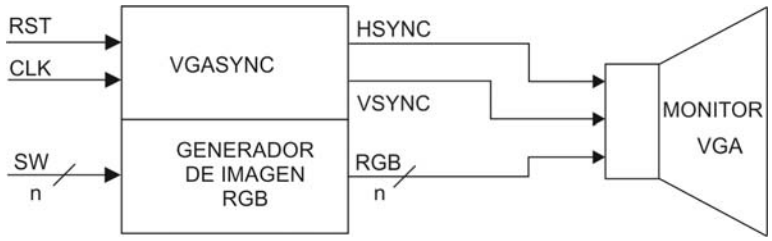


Figura 3.6 Diagrama General de Sincronía VGA

En la *figura 3.7* se muestra la descripción del módulo de sincronización VGA, el cual está formado por las descripciones, MOD 2, MOD 525, MOD 800 y COMPARADOR. Este sería nuestro segundo nivel de diseño en diagramas de bloques.

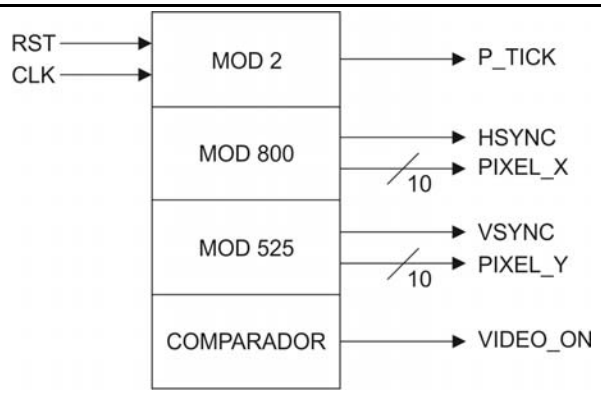


Figura 3.7 Diagrama del Circuito de Sincronía VGASYNC

Una vez comprendida la teoría de la sección 3.5, se procede a realizar el diseño de los diagramas a bloques de cada módulo que compone nuestro sistema principal, en este caso, el sistema mostrará los colores de la sección 3.3, con cualquiera de las tres tarjetas mencionadas anteriormente.

El circuito de la *figura 3.8*, muestra cómo se interconecta cada módulo que compone la sincronía de la tarjeta con él monitor.

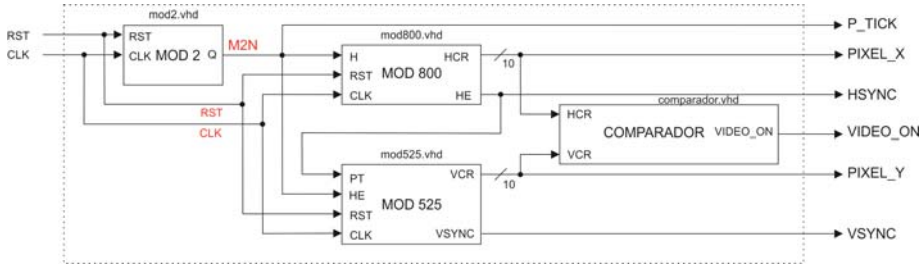


Figura 3.8 Circuito detallado de la Sincronía VGA

La *figura 3.8* se codifica en el *listado 3.1*, donde se interconectan los componentes que forman nuestro circuito de sincronía VGA.

Listado 3.1. Circuito de Sincronización VGA

```
1.library IEEE;
2.use IEEE.std_logic_1164.all;
3.use IEEE.std_logic_arith.all;
4.use IEEE.numeric_std.all;
5.entity VGASYNC is
6.  port(
7.  RST     : in  std_logic;
8.  CLK     : in  std_logic;
9.  P_TICK  : out std_logic;
10. VIDEO_ON: out std_logic;
```

```
11. HSYNC   : out std_logic;
12. VSYNC   : out std_logic;
13. PIXEL_X : out std_logic_vector(9 downto 0);
14. PIXEL_Y : out std_logic_vector(9 downto 0)
15. );
16. end VGASYNC;

17. architecture Sincronia of VGASYNC is
18. component FFD
19.         port(
20.         RST: in  std_logic;
21.         CLK: in  std_logic;
22.         Q  : out std_logic
23.         );
24. end component;

25. component Mod_NH
26.         port(
27.         RST: in  std_logic;
28.         CLK: in  std_logic;
29.         H  : in  std_logic;           -- Pixel Tick
30.         M1 : in  std_logic_vector(9 downto 0);
31.         M2 : in  std_logic_vector(9 downto 0);
32.         M3 : in  std_logic_vector(9 downto 0);
33.         HE : out std_logic;           -- Fin de la cuenta Horizontal
34.         HS : out       std_logic;
35.         HC : out std_logic_vector(9 downto 0)
36.         );
37. end     component;

38. component Mod_NV
39.         port(
40.         RST: in  std_logic;
41.         CLK: in  std_logic;
42.         S  : in  std_logic;           -- Pixel Tick
43.         HE : in  std_logic;
44.         M1 : in  std_logic_vector(9 downto 0);
45.         M2 : in  std_logic_vector(9 downto 0);
46.         M3 : in  std_logic_vector(9 downto 0);
47.         VS : out std_logic;
48.         VC      : out std_logic_vector(9 downto 0)
49.         );
50. end component;

51. component Comparador
52.         port(
53.         HCR: in  std_logic_vector(9 downto 0);
54.         VCR: in  std_logic_vector(9 downto 0);
55.         X  : in  std_logic_vector(9 downto 0);
56.         Y  : in  std_logic_vector(9 downto 0);
57.         Z  : out std_logic            -- Salida de encendido del video
58.         );
59. end component;

            -- Señales Intermedias
            -- Contador Modulo 2
60. signal M2N : std_logic;
61. signal HE  : std_logic;
62. signal HS  : std_logic;
63. signal VS  : std_logic;
            -- Contadores de Sincronia
64. signal HC  : std_logic_vector(9 downto 0);
65. signal VC  : std_logic_vector(9 downto 0);

66. begin

67. Modulo_01: FFD         port map (RST, CLK, M2N);
68. Modulo_02: Mod_NH      port   map   (RST,   CLK,   M2N,   "1100011111",   "1010010000",
    "1011101111", HE, HS, HC);
69. Modulo_03: Mod_NV      port  map  (RST,  CLK,  M2N,  HE,  "1000001100",  "0111101010",
    "0111101011", VS, VC);
```

```
70. Modulo_04: Comparador port map (HC, VC, "1010000000", "0111100000", VIDEO_ON);


71. P_TICK  <= M2N;
72. HSYNC   <= HS;
73. VSYNC   <= VS;
74. PIXEL_X <= HC;
75. PIXEL_Y <= VC;

76. end Sincronia;
```

El circuito Mod2 o Divisor de frecuencia únicamente nos sirve para ajustar la frecuencia de 50 MHz a 25 MHz, el cual, consiste de un Flip Flop tipo D y una compuerta NOT, la frecuencia dividida es enviada por la señal M2N, la cual sirve como una señal de habilitación para los contadores Módulo 800 y Módulo 525. La descripción de este circuito se muestra en la *figura 3.9.* En *listado 3.2* se muestra el código vhd de este circuito.

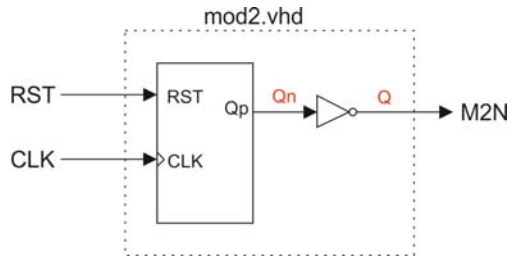


Figura 3.9 Divisor de Frecuencia MOD 2

Listado 3.2. Circuito Divisor de frecuencia Mod 2

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;

3. entity FFD is
4.      port(
5.         RST : in   std_logic;
6.         CLK : in   std_logic;
7.         Q   : out std_logic
8.       );
9. end FFD;

10. architecture Mod_2 of FFD is
11. signal Qp, Qn : std_logic;

12. begin
13.     Qn <= not Qp;

14.     RESET: process (RST, CLK, Qp)
15.     Begin
16.         if (RST = '0') then
17.             Qp <= '0';
18.         elsif (CLK'event and CLk = '1') then
19.             Qp <= Qn;
```

```
20.              end if;
21.     end process RESET;
22.     Q <= Qp;
23. end Mod_2;
```

En la *figura 3.10* se muestra la simulación donde se observa que la frecuencia de 50 MHz es reducida a 25 MHz.

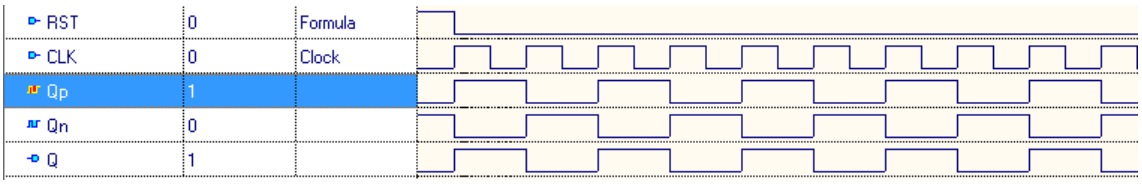


Figura 3.10 Simulación del Divisor de Frecuencia

En la *figura 3.11* se muestra el circuito del contador Módulo 800, el cual, es un contador ascendente de 0 a 799 ver con incrementos de 1 cada vez que la señal M2N habilita en estado alto al Flip Flop, entrega la señal HCR, el código de este circuito es mostrado en el *listado 3.3*.

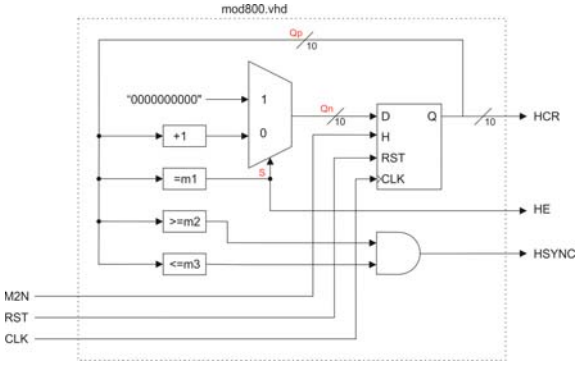


Figura 3.11 Circuito Contador Módulo 800

Listado 3.3. Circuito Contador Módulo 800

```
1.library IEEE;
2.use IEEE.std_logic_1164.all;
3.use IEEE.std_logic_arith.all;
4.use IEEE.std_logic_unsigned.all;

5.entity Mod_NH is
6.  port(
7.  RST  : in  std_logic;
8.  CLK  : in  std_logic;
9.  H    : in  std_logic;                -- Pixel Tick
10. M1   : in  std_logic_vector(9 downto 0);
11. M2   : in  std_logic_vector(9 downto 0);
12. M3   : in  std_logic_vector(9 downto 0);
```

```
13. HE   : out  std_logic;       -- Fin de la cuenta Horizontal
14. HS   : out     std_logic;
15. HC   : out std_logic_vector(9 downto 0)
16. );
17. end Mod_NH;

18. architecture Contador of Mod_NH is
19. signal S  : std_logic;       -- Fin de trazado HE o VE
20. signal Qn : std_logic_vector(9 downto 0);
21. signal Qp : std_logic_vector(9 downto 0);
22. begin

23.         SELECTOR: process (Qp, M1)
24.     begin
25.           if (Qp = M1) then
26.                S <= '1';
27.           else
28.                S <= '0';
29.           end if;
30.         end    process SELECTOR;

31.         MUX: process (S, Qp)
32.         begin
33.                case S is
34.                    when '0'    => Qn <= Qp + 1;
35.                    when others => Qn <= (others => '0');
36.                end case;
37.         end process MUX;

38.         HSYNC: process (Qp, M2, M3)
39.         begin
40.                if ((Qp >= M2) and (Qp <= M3)) then
41.                       HS <= '1';
42.                else
43.                       HS <= '0';
44.                end if;
45.         end process HSYNC;
46.
47.         RESET: process (RST, CLK, H, Qn, Qp)
48.         begin
49.                if (RST = '1') then
50.                       Qp <= (others => '0');
51.                elsif (CLK'event and CLK = '1') then
52.                       if (H = '1') then
53.                              Qp <= Qn;
54.                       end if;
55.                end if;
56.         end process RESET;
57.         HE <= S;
58.         HC <= Qp;

59. end Contador;
```

En la *figura 3.12* se muestra la simulación del contador módulo 800, cabe aclarar que no es posible mostrar la simulación completa así que solo se muestra una parte de la simulación. Sin embargo se realiza una simulación corta de cinco cuentas.

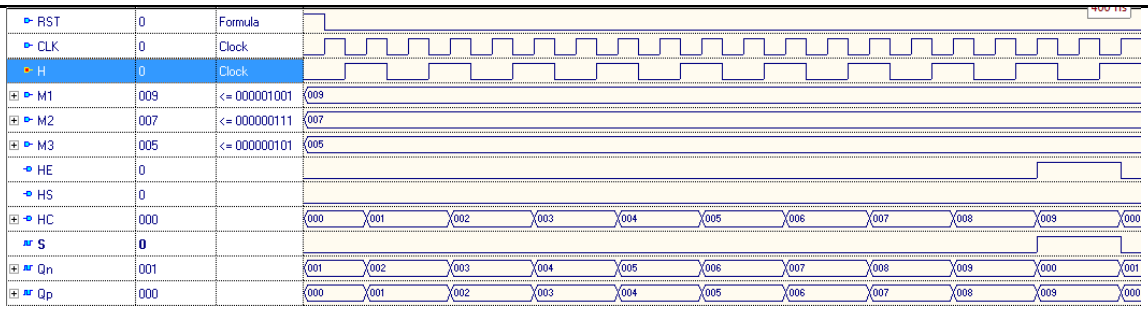


Figura 3.12 Simulación del Contador Módulo 800

En la *figura 3.13* se muestra el contador módulo 525 que al igual que el módulo de la figura 3.11, es un contador ascendente, con la excepción de que este contador nos indica el barrido vertical, es decir, cada que el contador Módulo 800 llega a su cuenta máxima envía una señal HE que habilita el registro del contador y comienza su cuenta. El código vhd del circuito Módulo 525 se encuentra en el *listado 3.4*.

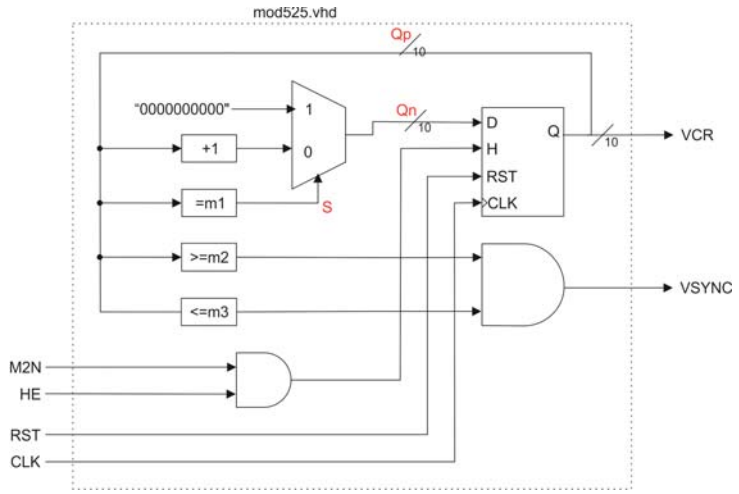


Figura 3.13 Circuito Contador Módulo 525

Listado 3.4. Circuito Módulo 525

```
1.library IEEE;
2.use IEEE.std_logic_1164.all;
3.use IEEE.std_logic_arith.all;
4.use IEEE.std_logic_unsigned.all;

5.entity Mod_NV is
6.  port(
7.  RST: in  std_logic;
8.  CLK: in  std_logic;
9.  S  : in  std_logic;          -- Pixel Tick
10. HE : in  std_logic;
11. M1 : in  std_logic_vector(9 downto 0);
12. M2 : in  std_logic_vector(9 downto 0);
13. M3 : in  std_logic_vector(9 downto 0);
14. VS : out       std_logic;
15. VC : out std_logic_vector(9 downto 0)
```

```
16. );
17. end Mod_NV;
18.
19. architecture Contador of Mod_NV is
20. signal S1 : std_logic;       -- Fin de trazado HE o VE
21. signal H  :    std_logic;
22. signal Qn : std_logic_vector(9 downto 0);
23. signal Qp : std_logic_vector(9 downto 0);
24. begin

25.         SELECTOR: process (Qp, M1)
26.         begin
27.                 if (Qp = M1) then
28.                         S1 <= '1';
29.                 else
30.                         S1 <= '0';
31.                 end if;
32.         end    process SELECTOR;

33.         MUX: process (S1, Qp)
34.         begin
35.          case S1 is
36.                  when '0'   => Qn <= Qp + 1;
37.                  when others => Qn <= (others => '0');
38.            end case;
39.         end process MUX;

40.         VSYNC: process ( Qp, M2, M3)
41.         begin
42.                 if ((Qp >= M2) and (Qp <= M3)) then
43.                         VS <= '1';
44.                 else
45.                         VS <= '0';
46.                 end if;
47.         end process    VSYNC;

48.         H <= S and HE;

49.         RESET: process (RST, CLK, H, Qn, Qp)
50.         begin
51.                 if (RST = '1') then
52.                         Qp <= (others => '0');
53.                 elsif (CLK'event and CLK = '1') then
54.                         if (H = '1') then
55.                                 Qp <= Qn;
56.                         end if;
57.                 end if;
58.         end process RESET;

59.         VC <= Qp;

60. end Contador;
```

El circuito comparador genera la señal **VIDEO_ON**, la cual permite exhibir la región deseada en el monitor mientras su valor sea ‘1’. El diagrama de bloques de este componente se muestra en la *figura 3.14*. El código vhd de este componente se encuentra en el *listado 3.5*.

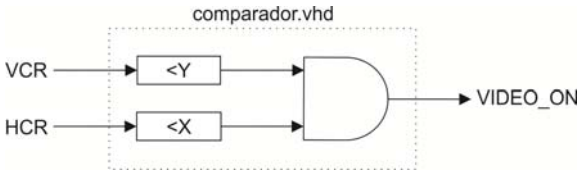


Figura 3.14 Circuito Comparador

Listado 3.5. Circuito Comparador

```
1.library IEEE;
2.use IEEE.std_logic_1164.all;
3.
4.entity Comparador is
5.  port(
6.  HCR: in  std_logic_vector(9 downto 0);
7.  VCR: in  std_logic_vector(9 downto 0);
8.  X  : in  std_logic_vector(9 downto 0);
9.  Y  : in  std_logic_vector(9 downto 0);
10. Z  : out std_logic    --Salida de encendido del video
11. );
12. end Comparador;
13.
14. architecture Comparador of Comparador is
15. begin

16.     COMPARACION: process (HCR, VCR, X, Y)
17.     begin
18.           if (HCR < X and VCR < Y) then
19.                 Z <= '1';
20.           else
21.          Z <= '0';
22.           end if;
23.        end process COMPARACION;

24. end Comparador;
```

Por último, se tiene un módulo que genera un color RGB en cada pixel dependiendo del valor de entrada por la señal **SW**, su diagrama es mostrado en la *figura 3.15* y su código en el *listado 3.6*. Con todo esto se tiene una sincronía con el monitor y, además, con este circuito podemos realizar y comprobar la prueba de la sincronía mostrando los 8 colores básicos que se vieron en las tablas de la sección 3.3. Éstos resultados se ilustran en el capítulo 6 de pruebas y resultados.

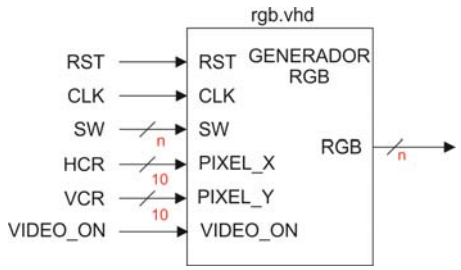


Figura 3.15 Circuito Generador RGB del Pixel

Listado 3.6. Código del Circuito Generador de RGB

```
1.Library IEEE;
2.use IEEE.std_logic_1164.all;

3.entity RGB_VGA is
4.      generic(
5.      n : integer := 8
6.      );
7.      port(
8.      RST    : in  std_logic;
9.      CLK    : in  std_logic;
10. SW     : in  std_logic_vector(n-1 downto 0);
11. P_TICK : out std_logic;
12. PIXEL_X: out std_logic_vector(9 downto 0);
13. PIXEL_Y: out std_logic_vector(9 downto 0);
14. HSYNC  : out   std_logic;
15. VSYNC  : out std_logic;
16. RGB      : out std_logic_vector(n-1 downto 0)
17. );
18. end RGB_VGA;

19. architecture arch of RGB_VGA is

20. component VGASYNC is
21.    port(
22.    RST     : in  std_logic;
23.    CLK    : in  std_logic;
24.    P_TICK  : out std_logic;
25.    VIDEO_ON: out std_logic;
26.    HSYNC   : out std_logic;
27.    VSYNC   : out std_logic;
28.    PIXEL_X : out std_logic_vector(9 downto 0);
29.    PIXEL_Y : out std_logic_vector(9 downto 0)
30.    );
31. end component;

32. signal video_on : std_logic;
33. signal rgb_reg  : std_logic_vector(n-1 downto 0);

34. begin

35. Modulo_01: VGASYNC port map (RST, CLK, P_TICK,          video_on, HSYNC, VSYNC,
    PIXEL_X, PIXEL_Y);

36. RESET: process (RST, CLK, SW)
37. begin
38.    if (RST = '1') then
39.       rgb_reg <= (others => '0')
40.    elsif(CLK'event and CLK ='1') then
41.       rgb_reg <= SW;
42.    end if;
43. end process RESET;
44. RGB <= rgb_reg when video_on = '1' else (others => '0');
45. end arch;
```

Una vez obtenido el controlador VGA, se lleva a cabo el estudio de procesamiento de imágenes para comprender los componentes de una imagen, los fundamentos y tratamientos o procesos que pueden ser aplicados en una PC convencional. Este caso de estudio se ve en el capítulo 4.

CAPÍTULO IV
PROCESAMIENTO DE IMÁGENES

4.1. INTRODUCCIÓN [18]

Gracias al tratamiento de las imágenes que generan es posible detectar zonas deforestadas, evolución de fenómenos meteorológicos, etc.

Aunque la distinción entre procesamiento y análisis de imágenes no es obvia de forma inmediata, el procesamiento de imágenes puede ser visto como una transformación de una imagen en otra imagen, es decir, a partir de una imagen, se obtiene otra imagen modificada. Por otro lado, el análisis es una transformación de una imagen en algo distinto a una imagen, en consecuencia, el análisis de imágenes es un determinado tipo de información representando una descripción o una decisión. En cualquier caso, las técnicas de análisis de imágenes digitales son aplicadas a imágenes que han sido procesadas previamente.

La rápida evolución de la tecnología de imágenes digitales, acompañado por una gran demanda en el mercado de cámaras y pantallas, presentan un significante cambio para los desarrolladores de dispositivos, quienes desean crear productos de alta calidad.

Sofisticados algoritmos de procesamiento de imágenes están disponibles, para llevar a cabo una extracción de imágenes del hardware de captura y exhibición. Pero su implementación está limitada por algunos factores como: la complejidad intrínseca del algoritmo, la presión para reducir el costo de los materiales, la necesidad para soportar una amplia variedad de formatos, y el frecuente requisito para personalizar un ambiente particular de los dispositivos [Tusch, 2006].

4.2. PROCESAMIENTO DIGITAL DE IMÁGENES [18]

Una imagen puede ser definida como una función de dos dimensiones, $f(x, y)$, donde x y y son coordenadas espaciales (del plano), y la amplitud de f en cualquier par de coordenadas (x, y), se llama la intensidad o el nivel de gris de la imagen en ese punto. Cuando todos los valores de x, y, y la amplitud f son cantidades finitas, discretas, llamamos a la imagen una *imagen digital.*

El campo del *Procesamiento Digital de Imágenes* se refiere al procesamiento digital de imágenes por medio de una computadora digital. Observe que una imagen digital está compuesta por un número finito de elementos, cada uno tiene una localización y valor particular. Estos elementos son conocidos como ***pixeles***. Pixel es el término ampliamente utilizado para denotar los elementos de una imagen digital.

4.3. ORÍGENES

Una de las primeras aplicaciones de las imágenes digitales fue en la industria periodística, cuando las primeras fotografías fueron enviadas por cable submarino entre Londres y Nueva York. La introducción en la década de 1920 de Bartlane, un sistema de transmisión por cable, redujo el tiempo de requerido para transportar una fotografía a través del Atlántico, de más de una semana a menos de tres horas.

Algunos problemas iniciales en la mejora de la calidad visas de las fotografías digitales, fueron relacionados con la selección de procedimientos de la impresión y la distribución de los niveles de intensidad.

Las primeras computadoras que alcanzaban potencialmente realizar tareas de procesamiento de imágenes aparecieron en los 60’s. El nacimiento de lo que hoy en día llamamos procesamiento digital de imágenes puede ser remontado a la disponibilidad de estas máquinas y al inicio del programa espacial durante ese período. Se combinó esos dos desarrollos para traer el inicio potencial del concepto de Procesamiento de imágenes Digitales, como se muestra en la *figura 4.1*.

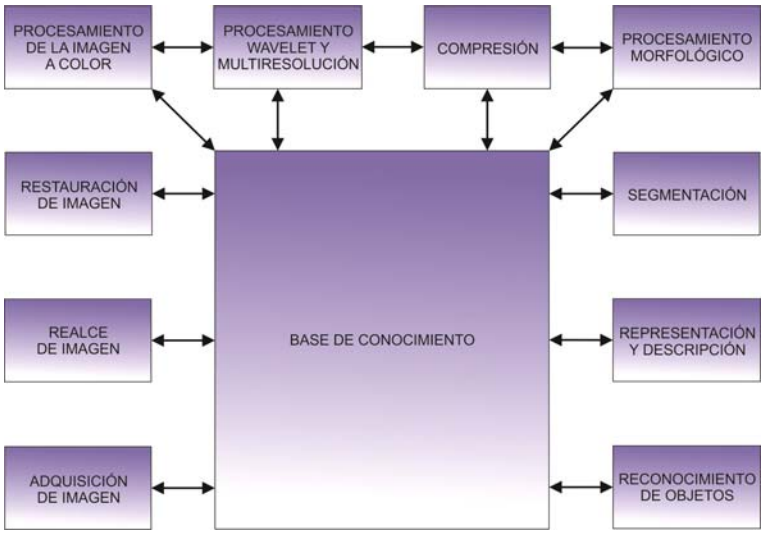


Figura 4.1. Pasos Fundamentales para el Procesamiento Digital de Imágenes

- ***Adquisición de Imagen***. La adquisición puede ser tan simple como lo es una imagen en forma digital ya dada. Generalmente, esta etapa implica el realizar un reprocesamiento, así como un escalamiento.

- ***Realce de Imagen***. Es una de las áreas más simples y atractivas del procesamiento digital de imágenes. Un ejemplo de realce es cuando se incrementa el contraste de una imagen porque "se ve mejor". Es un área de procesamiento de imágenes muy subjetiva. Se basa en preferencias subjetivas humanas con respecto qué constituye un "buen" resultado de realce.

- ***Restauración de Imagen***. Es un área que también se dedica a mejorar el aspecto de la imagen, a diferencia que está área es objetiva, en el sentido las técnicas de restauración tienden a ser basadas en modelos matemáticos o de probabilidad de la degradación de la imagen.

- ***Procesamiento de la Imagen a Color***. Es un área que ha ido ganando importancia debido al aumento significativo en el uso de imágenes digitales sobre el internet. El color es utilizado como base para extraer características del interés en una imagen.

- ***Procesamiento Wavelet y Multiresolución***. Es base para representar imágenes en varios grados de resolución. Particularmente este material se utiliza para la compresión de datos de imagen y representación piramidal, en la cual las imágenes se subdividen sucesivamente en regiones pequeñas.

- ***Compresión***. Se ocupa de las técnicas para reducir almacenaje requerido para guardar una imagen, o el ancho de banda requerido para transmitirla.

- ***Procesamiento Morfológico***. Trata de las herramientas para extraer los componentes de la imagen que son útiles en la representación y la descripción de la forma.

- ***Segmentación***. Los procedimientos de la segmentación reparten una imagen en sus componentes u objetos. La segmentación autónoma es generalmente una de las tareas más difíciles del procesamiento digital de imágenes.

- ***Representación y Descripción***. Casi siempre siguen la salida de la etapa de segmentación, que es generalmente datos crudos del pixel, el constituir cualquier límite de una región o de todos los puntos en la región de sí mismo. En cualquier caso, es conveniente convertir los datos a una forma que para el tratamiento es necesario para la computadora.

- ***Reconocimiento de Objetos***. Es el proceso que asigna una etiqueta a un objeto basado en sus descriptores.

- ***Base de Conocimiento***. El conocimiento sobre un dominio del problema se cifra en un sistema de tratamiento de la imagen bajo la forma de base de datos. Este puede estar como regiones de detalle de una imagen don de la información del interés se sabe para ser localizado, así limitando la búsqueda que tiene que ser conducida en buscar esa información.

4.4. MODELO DE FORMACIÓN DE UNA IMAGEN SIMPLE

Como ya se mencionó una imagen puede estar en función *f*(*x*, *y*), en la mayor parte de las imágenes antes de aplicar algún filtro, es necesario cambiar escala de color a escala de grises. Cuando una imagen se genera de un proceso físico, sus valores son proporcionales a la energía irradiada. Por lo cual *f*(*x*, *y*) debe ser diferente de cero e infinito, esto es:

$$0 < f(x, y) < \infty \qquad (4.1)$$

La función *f*(*x*, *y*) se puede caracterizar por dos componentes:

1. La cantidad incidente de la fuente de iluminación en la escena vista.
2. La cantidad de iluminación reflejada por los objetos en la escena.

Apropiadamente son llamados componentes de *iluminación* y *reflexión* y son denotados por *i*(*x*, *y*) y *r*(*x*, *y*), respectivamente. Las dos funciones combinan como producto para formar *f*(*x*, *y*).

$$\mathrm{f(x,y)} = \mathrm{i(x,y)r(x,y)} \qquad (4.2)$$

Dónde:

$$0 < \mathrm{i(x,y)} < \infty \qquad (4.3)$$

Y

$$0 < \mathrm{r(x,y)} < 1 \qquad (4.4)$$

La ecuación *(4.4)* indica que la reflexión es limitada por 0 (absorción total) y 1 (reflexión total. La naturaleza de *i*(*x*, *y*) es determinada por la fuente de iluminación, y *r*(*x*, *y*) es determinado por las características de los objetos reflejados.

4.5. REPRESENTANDO IMÁGENES DIGITALES

El resultado del muestreo y de la cuantificación es una matriz de números reales. Asumiendo que una imagen es $f(x, y)$ es muestreada y que el resultado de la imagen digital tiene M renglones y N columnas. Los valores de las coordenadas (x, y) son convertidas en cantidades *discretos*. Se usan valores enteros para éstas coordenadas discretas, así los valores de las coordenadas en el origen son $(x, y) = (0, 0)$. Los valores de coordenadas del siguiente renglón de la imagen son representados por (0, 1). Es importante recordar que la notación (0, 1) es usado para representar la segunda muestra a lo largo de la primera fila, no significa que éstos son los valores reales de las coordenadas físicas cuando la imagen es muestreada. La *figura 4.2* muestra las coordenadas como serán usadas en este proyecto.

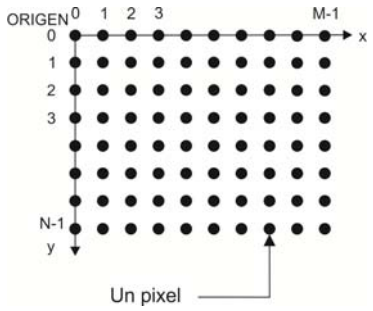


Figura 4.2. Coordenadas de los Pixeles

4.6. FUNDAMENTOS DE LAS IMÁGENES DIGITALES

Son el principal ingrediente de lo que se conoce como visión por computador y representan mediante algún tipo de codificación, normalmente en una matriz de números de dos dimensiones, una escena del entorno.

Las imágenes de intensidad miden la cantidad de luz que incide en un dispositivo fotosensible, mientras que las imágenes de alcance estiman directamente la estructura en tres dimensiones (3D) de la escena. Un ejemplo típico de una imagen de intensidad es una fotografía, mientras que de una imagen de alcance es, por ejemplo, la imagen que obtiene el oftalmólogo sobre el grado de rugosidad de la córnea de un paciente o las imágenes de radar.

En definitiva e independiente del tipo de sensor utilizado, la imagen que ha de ser tratada por la computadora se presenta digitalizada especialmente en forma de matriz con una resolución de *m x n* elementos. Cada elemento de la matriz o *píxel* tendrá un valor asignado que corresponde con el nivel de luminosidad del punto correspondiente en la escena captada; dicho valor es el resultado de la cuantificación de *intensidad* o *nivel de gris*. Los términos intensidad y nivel de gris se suelen usar indistintamente.

4.6.1. Imagen a Color

Los elementos de la matriz, vienen dados por tres valores, que representan cada uno de los componentes básicos del color en cuestión. Estos componentes son el *Rojo (R), Verde (G) y Azul (B),* el conocido código *RGB*, en este caso el conjunto de valores (0,0,0) es el negro absoluto; el (255,255,255) el blanco absoluto; el (255, 0, 0) el rojo puro; (0,255,0) el verde puro; y el (0,0,255) el azul puro. Como es lógico, la combinación de distintos valores proporciona otros colores. Debido a lo anterior, se dirá que una imagen en color posee tres bandas espectrales, rojo, verde y azul; cada una de ellas es una matriz de números 2D con valores en el rango 0 a 255 para imágenes de 8 bits. A veces para representar una imagen digital a color, se utiliza una representación basada en una matriz de valores y un mapa de colores o paleta asociada.

4.6.2. Escala de Grises

En una imagen en escala de grises el pixel posee un valor equivalente a una intensidad de gris, que va desde el negro más profundo variando gradualmente la intensidad de gris hasta llegar al blanco. Éstas imágenes se representan con 8 bit para representar cada pixel lo que nos permite únicamente obtener 256 tonalidades diferentes (0 a 255). Cuando se convierte una imagen a color a escala de grises se debe obtener el promedio de las 3 bandas de color de cada pixel. Esta operación se muestra en la *ecuación (4.5)*.

$$Gray(x,y) = (f(x,y,R) + f(x,y,G) + f(x,y,B))/3 \qquad (4.5)$$

4.6.3. Imágenes en Blanco/Negro o Binarización

Si la imagen es en *Blanco y Negro* **(B/N)**, se almacena un valor por cada píxel, este valor es el nivel de intensidad o nivel de gris comentado anteriormente. Se suele utilizar un rango de valores para su representación, que generalmente es de 0 a 2^{n-1}.

Para obtener una imagen en blanco y negro, debemos obtener primero la imagen en escala de grises, y establecemos un valor umbral, con el cual vamos a definir en qué momento tendremos un valor negro o un valor blanco, este umbral se representa en la *ecuación (4.6).*

$$BW(x,y,z) = \begin{cases} 0 \; si \; Gray(x,y) \leq \lambda \\ 1 \; si \; Gray(x,y) \geq \lambda \end{cases} \qquad (4.6)$$

Donde λ representa el valor del umbral deseado

Cabe mencionar que para aplicar un filtro o proceso a una imagen, primeramente habrá que aplicar una conversión a la imagen a color a imagen en escala de grises y enseguida aplicar el filtro deseado.

4.7. EXTRACCIÓN DE BORDES, ESQUINAS Y PUNTOS DE INTERÉS [1]

Los *puntos de borde*, o simplemente *bordes* son píxeles alrededor de los cuales la imagen presenta una brusca variación en los niveles de gris. El objetivo consiste en dada una imagen, que puede o no estar corrompida por ruido, localizar los bordes más probables generados por elementos de la escena y no por ruido.

En realidad el borde se refiere a cadenas conectadas de punto de borde, esto es fragmentos de contorno, esto no impide que la imagen también pueda contener puntos aislados que presentan un alto contraste en los niveles de gris. Los puntos de borde a veces

son denominados "*edgels*" (procedente de *edge elements*). Existen varias razones que sostienen el interés por los bordes. Los contornos de los objetos sólidos de la escena, las marcas en las superficies, las sombras, todas generan bordes. Además las líneas de las imágenes, las curvas y contornos son características o elementos básicos para muchas aplicaciones tales como calibración, movimiento o reconocimiento.

La detección de bordes es una parte fundamental de la mayoría de sistemas de visión puesto que el éxito de los niveles siguientes de procesamiento depende fuertemente de la fiabilidad de las características, en este caso bordes. La filosofía básica de muchos algoritmos de detección de bordes es el cómputo de operadores derivada locales (primera o segunda). Este trabajo se enfoca a dejar una base del cómo se puede implementar un filtro en FPGA, siendo el filtro de estudio el **Gradiente**, el cual se ve en la *sección 4.7.1*.

4.7.1. Gradiente de una Imagen [1]

El gradiente de una imagen *f(x, y)* en un punto *(x, y)* se define como un vector bidimensional dado por la *ecuación (4.7)*, siendo un vector perpendicular al borde.

$$G[f(x,y)] = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \frac{d}{dx} f(x,y) \\ \frac{d}{dy} f(x,y) \end{bmatrix} \qquad (4.7)$$

Donde el vector ***G*** apunta a la dirección de variación máxima de *f* en el punto *(x, y)* por unidad de diferencia con la magnitud y dirección dadas por

$$|G| = \sqrt{G_x^2 + G_y^2}; \qquad \emptyset(x,y) = \tan^{-1}\frac{G_y}{G_x} \qquad (4.8)$$

Es una práctica habitual aproximar la magnitud del gradiente con valores absolutos

$$|G| \approx |G_x| + |G_y| \qquad (4.9)$$

Esto se hace por que el valor de la magnitud del gradiente no es tan importante como la relación entre diferentes valores. Es decir, se va a decidir si un punto es de borde según que la magnitud del gradiente supere o no un determinado umbral, pues bien sólo es cuestión de ajustar dicho umbral para que el resultado de la extracción de bordes sea el mismo, tanto si se calcula la magnitud del gradiente mediante *(4.8)* como si se hace mediante *(4.9)* y sin embargo, esta última ecuación es mucho más fácil de implementar, particularmente cuando se realiza en hardware.

Para calcular la derivada en *(4.7)* se pueden utilizar las diferencias de primer orden entre dos pixeles adyacentes; esto es,

$$G_x = \frac{f(x+\Delta x)-f(x-\Delta x)}{2\Delta x} \qquad G_y = \frac{f(y+\Delta y)-f(y-\Delta y)}{2\Delta y} \tag{4.10}$$

Ésta es la forma más elemental de obtener el gradiente en un punto. La magnitud del gradiente puede obtener cualquier valor real y ángulo también cualquier valor real entre 0° y 360°. No obstante hay otros operadores ampliamente difundidos para implementar el concepto de derivada en un punto y que consideran una vecindad de dimensión 3x3 entorno al punto, como lo es el operador de *Sobel*.

Siguiendo el concepto de derivada definido en *(4.10)*, se puede obtener una imagen binarizada considerando un umbral de *T* para la magnitud del gradiente, es decir

$$g(x,y) = \begin{cases} 1 \; si \; G[f(x,y)] > T \\ 0 \; si \; G[f(x,y)] \leq T \end{cases} \tag{4.11}$$

La *figura 4.3* muestra la imagen gradiente obtenida en MATLAB, y el *listado 4.1* muestra el algoritmo realizado para obtener dicha imagen.

Figura 4.3. Gradiente de imagen

Listado 4.1 Gradiente de Imagen

```
1.  Gx = 0;
2.  Gy = 0;
3.  for m=2 : ax - 1
4.    for n=2 : ay – 2
5.      for o = 1 : az
6.        Gx = ( img_gray(m-1, n+2, o) + ...
7.             2*img_gray(m  , n+1, o) + ...
8.               img_gray(m+1, n-1, o))- ...
9.             ( img_gray(m-1, n-1, o) + ...
10.       2*img_gray(m  , n-1, o) + ...
11.         img_gray(m+1, n-1, o));
12.  Gy = ( img_gray(m+1, n-1, o) + ...
13.       2*img_gray(m+1, n  , o) + ...
14.         img_gray(m+1, n+1, o))- ...
15.       (img_gray(m-1, n-1, o) + ...
16.       2*img_gray(m-1, n  , o) + ...
17.       img_gray(m-1, n+1, o));
18.  G = abs(Gx)+abs(Gy);
19.  if (G > .505)
20.     img_grad(m, n, o)=1;
21.  else
22.     img_grad(m, n, o)=0;
23.  end;
24. end;
25. end;
26. end;
```

Una vez estudiado la parte de las imágenes se procede a desarrollar los algoritmos que se utilizan para exhibir imágenes en FPGA, esta sección se ve en el capítulo 5.

CAPÍTULO V IMPLEMENTACIÓN DEL SISTEMA EN FPGA

5.1. LECTURA DE UNA IMAGEN

En este capítulo se proponen dos metodologías para la lectura de imágenes en FPGA, desde pasar la imagen al FPGA hasta como mostrarla a través del monitor por medio de la tarjeta de desarrollo utilizada, los diagramas de flujo utilizados en MATLAB y los diagramas a bloques diseñados para cada metodología propuesta. Por último, se deja una base para la implementación de filtros como el gradiente.

Las metodologías propuestas son basadas en dos tipos de memoria que son:

- ROM
- RAM

5.1.1. Memoria ROM

En esta sección se implementa una imagen en FPGA como memoria ROM, esto se logra realizando un programa en MATLAB, para generar un archivo vhd al cual le daremos el nombre de **ROM_FIGURE**, en el cual se tiene toda la información de la imagen, como el número del pixel y sus valores en RGB. En la *figura 5.1* se muestra un diagrama de flujo del proceso a seguir para realizar la conversión de la imagen a un archivo descriptivo de hardware.

Para la implementación del proceso de la *figura 5.1*, se utiliza MATLAB, por las facilidades que presenta en el manejo de las imágenes. El primer paso, es asignar la imagen a una matriz M. la matriz M generada, es de 3 dimensiones, por lo que, en el segundo bloque, el valor obtenido para *j* es el número de renglones, para *k* el número de columnas, y para *l* es de 3, pues la matriz es tridimensional.

En seguida, es necesario hacer el ajuste de los valores de la imagen M de 24 bit, para poder obtener una imagen N de 8 bit. Para esto, se hace uso de dos ciclos anidados con las

variables *r* (número de renglón), y *c* (número de columnas). Cada valor RGB de la matriz M debe ser dividido entre 255, para normalizar el valor del pixel, y posteriormente se debe multiplicar por 7, para ajustar a 3 bit, y por 3 para ajustar a dos bit. Tales ecuaciones se muestran en la *figura 4*. Finalmente, el último bloque representa la generación de una tabla de acceso a datos (LUT), que también, en cierta forma, se puede interpretar como una memoria ROM interna en el FPGA [19].

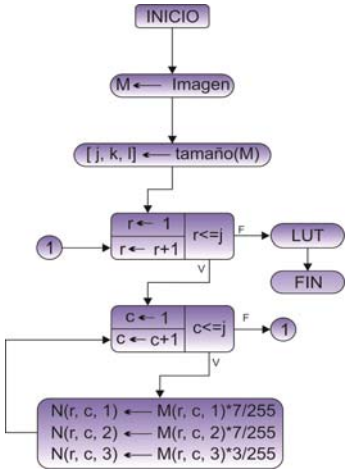


Figura 5.1. Diagrama de flujo del proceso de conversión de una imagen a lenguaje descriptivo de hardware.

Para generar el archivo descriptivo de hardware (LUT), de la *figura 5.1* se debe seguir el siguiente algoritmo:

1. Crear un archivo texto con el nombre de la imagen, con extensión *vhd*.

2. Se graba al archivo el encabezado de la descripción.

3. Determinar el número de bit para la dirección de datos, de acuerdo al tamaño de la matriz. Por lo pronto, nos referiremos al número de bit como *n*. se está considerando una imagen donde el número de renglones es igual al número de columnas, y el tamaño es potencia de dos.

4. La dirección de entrada a la LUT se forma con el número de renglón en la parte más significativa y el número de columna en la parte menos significativa.
5. Se envía al archivo la entidad, incluyendo los puertos de dirección, y los datos de salida para el video.

6. Se genera la sección de arquitectura, con su begin.

7. Se genera el proceso, donde la dirección es utilizada en un proceso *case*.

8. Se convierte el valor de la dirección a una cadena que contiene el valor en binario, así como el dato de salida, que debe convertirse igualmente a binario, en otra cadena.

9. Se genera cada uno de los casos utilizando las cadenas con las conversiones a binario, y se almacenan en el archivo texto, cuidando la sintaxis del lenguaje descriptivo de hardware.

10. Se cierra el proceso.

11. Se cierra la arquitectura.

12. Se agrega la descripción a un módulo de mayor jerarquía.

Los pasos descritos anteriormente nos llevan al código que se muestra en el *listado 5.1*.

Listado 5.1. Código MATLAB para generar el archivo rom_figure.vhd

```
1    close all;
2    clear all;
3    clc;
4    Figura=imread('clio256.jpg');
5    [j,k,l]=size(Figura);
6    RGB=double(zeros(j*k,1));
     %%PARA IMÁGENES A COLOR
7    for r=1:j,
8        for c=1:k,
9             R=(double(Figura(r,c,1))/255)*7;
10            G=(double(Figura(r,c,2))/255)*7;
11            B=(double(Figura(r,c,3))/255)*3;
12            indice=j*(r-1)+c;
```

```
         RGB(indice) = uint8(R)*32 + uint8(G)*4 + uint8(B);
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%             Binarización                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Fig=uint8(zeros(j*k,8));
%Binarización del Índice
m=16;
for r=1:j*k,
    Auxiliar = r-1;
    for i=m:-1:1,
        IDX(r,i) = uint8(rem(Auxiliar,2));
        Auxiliar = floor(Auxiliar/2);
    end;
end;
%Binarización del Dato
% número de bits
n=8;
for r=1:j*k,
    Auxiliar = RGB(r);
    for i=n:-1:1,
        DAT(r,i) = uint8(rem(Auxiliar,2));
        Auxiliar = floor(Auxiliar/2);
    end;
end;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Generación Automática del archive VHDL de los coeficientes de la figura ROM %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Se Abre el archivo VHDL
fid = fopen('ROM_FIGURE.vhd','wt');
%Descripción del encabezado
fprintf(fid,'-- FIGURA RGB 8 bit\n');
fprintf(fid,'-- Facultad de Informática - UAQ\n');
fprintf(fid,'--\n');
fprintf(fid,'-- Carlos Alberto Ramos Arreguín\n');
fprintf(fid,'--\n');
fprintf(fid,'\n');
fprintf(fid,'library IEEE;\n');
fprintf(fid,'use IEEE.std_logic_1164.all;\n');
fprintf(fid,'\n');
fprintf(fid,'entity ROM_FIGURE is\n');
fprintf(fid,'   port(\n');
% Declaración del coeficiente del Índice
fprintf(fid,'      I : in  std_logic_vector(%d downto 0);\n',m-1);
%Declaración del coeficiente de RGB
fprintf(fid,'      A : out std_logic_vector(%d downto 0)\n',n-1);
fprintf(fid,'      );\n');
fprintf(fid,'   end ROM_FIGURE;\n');
fprintf(fid,'\n');
fprintf(fid,'architecture LUTable of ROM_FIGURE is\n');
fprintf(fid,'begin\n');
fprintf(fid,'   process(I)\n');
fprintf(fid,'   begin\n');
fprintf(fid,'      case I is\n');
fprintf(fid,'         -- Coefficient RGB format 4 bit\n');
%Generación Automática de la Tabla
for i=1:j*k
    fprintf(fid,'         when "');
    %Coeficiente del índice en binario
    for j=1:m
        fprintf(fid,'%d',IDX(i,j));
    end;
    fprintf(fid,'" => A <= "');
    %Coeficiente del dato en binario
    for j=1:n
        fprintf(fid,'%d',DAT(i,j));
    end;
    %Coeficiente del Índice en Decimal
    fprintf(fid,'"; -- Index %d   Coefficient %d\n',i-1,RGB(i));
end;
%Fin de la Tabla
```

```
67  fprintf(fid,'         when others => A <= "');
68  for i=1:n
69      fprintf(fid,'0');
70  end;
71  fprintf(fid,'"; -- Índices Irrelevantes\n');
72  fprintf(fid,'      end case;\n');
73  fprintf(fid,'   end process;\n');
74  fprintf(fid,'end LUTable;\n');
75  fclose(fid);
```

Para la implementación en hardware se diseña un diagrama de bloques como el de la figura 2. Utilizando la arquitectura del capítulo 3 para la sincronía con el monitor y la generación de los valores de cada pixel. El código del sistema de la *figura 5.2* se muestra en el *listado 5.2, 5.3 y 5.4.*

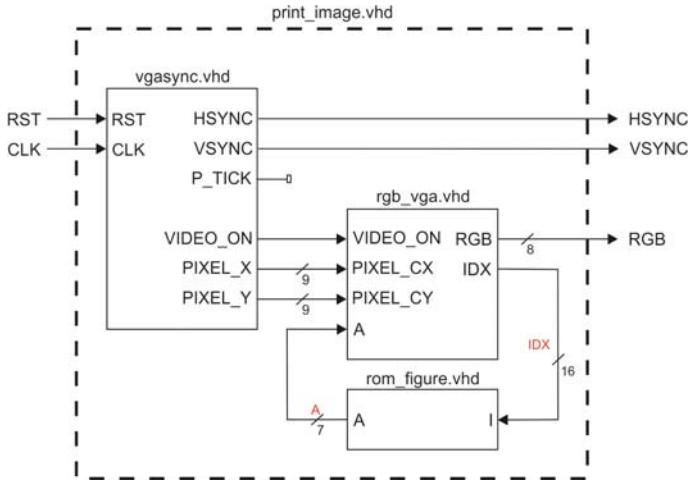


Figura 5.2. Diagrama a bloques del sistema de Imagen ROM.

Listado 5.2 Código del sistema Print Image como ROM

```
1.Library IEEE;
2.use IEEE.std_logic_1164.all;

3.entity Print_Image is
4.    port(
5.    RST: in  std_logic;
6.    CLK: in  std_logic;
7.    HSYNC    : out std_logic;
8.    VSYNC    : out std_logic;
9.    RGB_IMAGE: out std_logic_vector(7 downto 0)
10. );
11. end Print_Image;

12. architecture print of Print_Image is

13. component VGASYNC is
14.    port(
15.    RST     : in  std_logic;
16.    CLK  : in  std_logic;
17.    P_TICK  : out std_logic;
18.    VIDEO_ON: out std_logic;
19.    HSYNC       : out std_logic;
```

```
20.    VSYNC        : out std_logic;
21.    PIXEL_X : out std_logic_vector(9 downto 0);
22.    PIXEL_Y : out std_logic_vector(9 downto 0)
23. );
24. end    component;

25. component RGB_VGA is
26.    port(
27.    VIDEO_ON: in  std_logic;
28.    PIXEL_CX: in  std_logic_vector( 9 downto 0);
29.    PIXEL_CY: in  std_logic_vector( 9 downto 0);
30.    A            : in  std_logic_vector(7 downto 0);
31.    IDX   : out std_logic_vector(15 downto 0);
32.    RGB   : out std_logic_vector(7 downto 0)
33.    );
34. end component;

35. component ROM_FIGURE is
36.    port(
37.    I: in  std_logic_vector( 15 downto 0);
38.    A: out std_logic_vector( 7 downto 0)
39.    );
40. end component;

41. signal VIDEO_ON  : std_logic;
42. signal PIXEL_TICK: std_logic;
43. signal PIXEL_X   : std_logic_vector( 9 downto 0);
44. signal PIXEL_Y   : std_logic_vector( 9 downto 0);
45. signal IDX       : std_logic_vector(15 downto 0);
46. signal A         : std_logic_vector(7 downto 0);

47. begin

48.       Modulo_01: VGASYNC port map (RST, CLK, PIXEL_TICK, VIDEO_ON,  HSYNC, VSYNC,
    PIXEL_X, PIXEL_Y);
49.       Modulo_02: RGB_VGA        port  map  (VIDEO_ON,  PIXEL_X,  PIXEL_Y,  A,  IDX,
    RGB_IMAGE);
50.       Modulo_03: ROM_FIGURE port map (IDX, A);

51. end print;
```

Listado 5.3. Código del Módulo RGB VGA

```
1.Library IEEE;
2.use IEEE.std_logic_1164.all;

3.entity RGB_VGA is
4.     port(
5.     VIDEO_ON: in  std_logic;
6.     PIXEL_CX: in  std_logic_vector( 9 downto 0);
7.     PIXEL_CY: in  std_logic_vector( 9 downto 0);
8.     A       : in  std_logic_vector( 7 downto 0);
9.     RGB     : out std_logic_vector( 7 downto 0);
10. IDX     : out std_logic_vector(15 downto 0)
11. );
12. end RGB_VGA;

13. architecture RGB of RGB_VGA is

14. signal S        : std_logic;
15. signal CRX1: std_logic_vector(1 downto 0);
16. signal CRY1: std_logic_vector(1 downto 0);
17. signal CRX2: std_logic_vector(7 downto 0);
18. signal CRY2: std_logic_vector(7 downto 0);

19. begin
20.      CRX1 <= PIXEL_CX(9 downto 8);   --  2  Bit  más  significativos  del  barrido
    horizontal
```

```
21.      CRY1 <= PIXEL_CY(9 downto 8);   --  2  Bit  más  significativos  del  barrido
    vertical
22.      CRX2 <= PIXEL_CX(7 downto 0);   --  5  Bit  menos  significativos  del  barrido
    horizontal
23.      CRY2 <= PIXEL_CY(7 downto 0);   --  5  Bit  menos  significativos  del  barrido
    vertical

24.      S <= CRY1(1) or CRY1(0) or CRX1(1) or CRX1(0);
25.      IDX <= CRY2 & CRX2;     -- Índice CRY2 indica los 5 Bit más significativos y
    CRX2 los 5 Bit menos significativos

26. RGB_FIGURE: process (VIDEO_ON, S, A)
27. begin
28.    if (VIDEO_ON = '0') then
29.      RGB <= (others => '0');
30. else
31.    if (S = '0') then
32.      RGB <= A;
33.    else
34.      RGB <= (others => '0');
35.    end if;
36. end if;

37. end process RGB_FIGURE;

38. end RGB;
```

En el *listado 5.4* se observa que en las líneas 19 a 21 se tiene * esto en lugar de puntos suspensivos, lo cual nos indica que el valor sigue continuo hasta llegar al final, esto, debido a que el total de líneas de este código supera las 65535 líneas y es inapropiado agregarlo completo en este documento del proyecto.

Listado 5.4. Código del Módulo ROM_FIGURE

```
1.library IEEE;
2.use IEEE.std_logic_1164.all;

3.entity ROM_FIGURE is
4.    port(
5.    I : in  std_logic_vector(15 downto 0);
6.    A : out std_logic_vector(7 downto 0)
7.    );
8.end ROM_FIGURE;

9.architecture LUTable of ROM_FIGURE is
10. begin
11.    process(I)
12.    begin
13.    case I is
    --Coefficiente RGB formato 3 bit R y G 2 bit B
14.      when "0000000000000000" => A <= "00000000"; -- Index 0   Coefficient 0
15.      when "0000000000000001" => A <= "00000000"; -- Index 1   Coefficient 0
16.      when "0000000000000010" => A <= "00000000"; -- Index 2   Coefficient 0
17.      when "0000000000000011" => A <= "00000000"; -- Index 3   Coefficient 0
18.      when "0000000000000100" => A <= "00000000"; -- Index 4   Coefficient 0
19. *
20. *
21. *
22.      when "1111111111111101" => A <= "11111111"; -- Index 65533   Coefficient 255
23.      when "1111111111111110" => A <= "11111111"; -- Index 65534   Coefficient 255
24.      when "1111111111111111" => A <= "11111111"; -- Index 65535   Coefficient 255
25.      when others => A <= "00000000"; -- Irrelevant indexes
26. end case;
```

```
27. end process;
28. end LUTable;
```

Una vez realizado lo anterior, ya puede presentar en un monitor la imagen ajustada a 8 bit. El resultado de esta prueba se encuentra en la figura 6.1 del siguiente capítulo.

5.1.2. MEMORIA SRAM

A diferencia del punto anterior, donde la imagen es almacenada en el FPGA como una tabla, los datos de la imagen son transferidos de la PC a la tarjeta de experimentación vía puerto serie o RS232 y almacenados en la memoria SDRAM de la misma. En este punto se propone una nueva metodología para llevar a cabo la exhibición de una imagen en un monitor, esta se muestra en la *figura 5.3*.

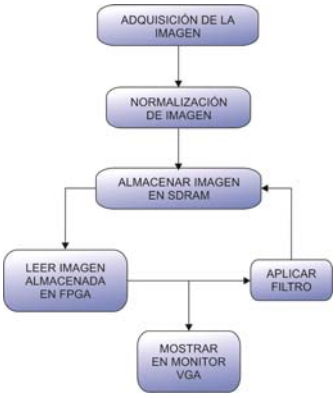


Figura 5.3. Metodología Propuesta de Procesamiento de Imágenes en Hardware

Para realizar lo anterior es necesario seguir los siguientes pasos:

1. Primero que nada establecer una comunicación serial entre nuestra PC con la tarjeta NEXYS 2, el lenguaje utilizado en este caso fue C#. el código utilizado es el del listado 5.5. En la figura 5.4 se ilustra cómo queda la interfaz de nuestro programa para la transferencia de los datos de la PC a la tarjeta de experimentación.

2. Utilizando el mismo diagrama de flujo de la figura 5.1, generamos nuestro archivo con los coeficientes de cada pixel, en este caso ya no es necesario hacer una conversión binaria de cada uno de ellos, por lo tanto, el código del listado 1 utilizado es acortado y ajustado, esto se muestra en el código del *listado 3*.

3. Una vez realizado lo anterior se programa nuestro FPGA, y el resultado obtenido se muestra en la *figura 5*.

Listado 5.5. Comunicación serial en C#

```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.ComponentModel;
4.  using System.Data;
5.  using System.Drawing;
6.  using System.Linq;
7.  using System.Text;
8.  using System.Windows.Forms;
9.  using System.IO;
10. namespace WindowsFormsApplicationSerialCom
    {
11.     public partial class Form1 : Form
        {
12.         public Form1()
            {
13.             InitializeComponent();
            }
14.         private void Enviar_Click(object sender, EventArgs e)
            {
15.             string s;
16.             byte[] valor2={0,1};
17.             s = CajaEnvio.Text;
18.             valor2[0] = Convert.ToByte(s);
19.             SerialPort.Write(valor2,0,1);
            }
20.         private void Form1_Load(object sender, EventArgs e)
            {
21.             Timer.Enabled = false;
            }
22.         private void Cerrar_Click(object sender, EventArgs e)
            {
23.             Timer.Enabled = false;
24.             SerialPort.Close();
25.             Close();
            }
26.         private void Timer_Tick(object sender, EventArgs e)
            {
27.             int j,i;
28.             j = SerialPort.BytesToRead;
29.             if (j > 0)
                {
30.                 i = SerialPort.ReadByte();
31.                 label1.Text = j.ToString();
32.                 CajaRecibo.Text = i.ToString();
                }
            }
33.         private void button1_Click(object sender, EventArgs e)
           {
34.             SerialPort.PortName = "COM"+PuertoCOM.Text;
35.             SerialPort.Open();
36.             Enviar.Enabled = true;
37.             Cerrar.Enabled = true;
38.             button2.Enabled = true;
```

```
39.             Timer.Enabled = true;
        }
40.         private void button2_Click(object sender, EventArgs e)
            {
41.             double j;
42.             progressBar1.Visible = true;
43.             using (StreamReader sr = new StreamReader ("Filename.txt"))
                {
44.                 string line;
45.                 j = 1;
46.                 byte[] valor2 = { 0, 1 };
47.                 CajaEnvio.Text = "";
48.                 Do
                    {
49.                     line = sr.ReadLine();
50.                     if (line != "")
                        {
51.                         CajaEnvio.Text = line;
52.                         valor2[0] = Convert.ToByte(line);
53.                         SerialPort.Write(valor2, 0, 1);
54.                         j = j + 1;
55.                         label1.Text = j.ToString();
                        }
                    }
56.                 while (j < 65536);
57.                 MessageBox.Show ("Fin del archivo", "Archivo", MessageBoxButtons.OK,
    MessageBoxIcon.Asterisk);
                }
            }
58.         private void CajaRecibo_TextChanged(object sender, EventArgs e)
            {
            }
59.         private void BtnOpen_Click(object sender, EventArgs e)
            {
60.             if (openFileDialog1.ShowDialog() == DialogResult.OK)
                {
61.                 System.IO.StreamReader sr = new System.IO.StreamReader
    (openFileDialog1.FileName);
62.                 MessageBox.Show(sr.ReadToEnd());
63.                 sr.Close();
                }
            }
64.         private void progressBar1_Click(object sender, EventArgs e)
            {
65.             for (int i = 0; i <= 4; i++)
                {
                    // Inserts code to copy a file
66.                 progressBar1.PerformStep();
                    // Updates the label to show that a file was read.
67.                 label1.Text = "# of Files Read = " + progressBar1.Value.ToString();
                }
            }
68.         private void label1_Click(object sender, EventArgs e)
            {
            }
        }
```

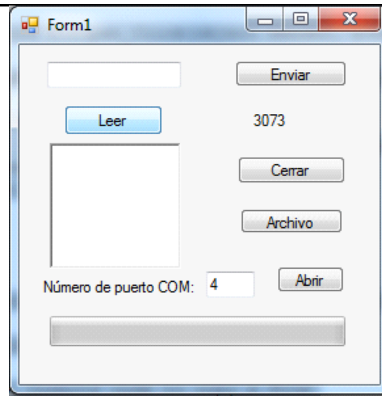


Figura 5.4. Interfaz para la transmisión de datos a la tarjeta de experimentación.

El código del *listado 5.1* es modificado y recortado obteniendo el código del *listado 5.6.*

Listado 5.6. Código de para generar el archivo .txt con los coeficientes de la imagen

```
close all;
clear all;
clc;
Figura=imread('Clio.bmp');
[j,k,l]=size(Figura);
RGB=double(zeros(j*k,1));
for r=1:j,
    for c=1:k,
        R=(double(Figura(r,c,1))/255)*7;
        G=(double(Figura(r,c,2))/255)*7;
        B=(double(Figura(r,c,3))/255)*3;
        indice=j*(r-1)+c;
    End
End
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Generación Automática del archive TXT de los coeficientes de la figura ROM
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fid = fopen('ROM_FIGURE.txt','wt');
for i=1:j*k
    %Valores Decimales
    fprintf(fid,'%d\n',i-1,RGB(i));
end;
%Close file
fclose(fid);
```

El diagrama de bloques para esta metodología también es modificado, ya que se agrega el bloque de interfaz RS232 para la recepción de los datos, y el bloque de interfaz con la memoria SDRAM, este diagrama se muestra en la *figura 5.4.*

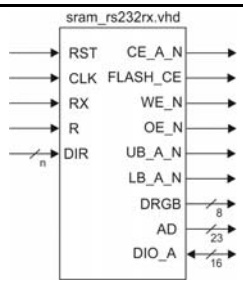


Figura 5.5. Diagrama de Bloques de la interfaz RS232 a SDRAM

Un diseño más detallado de esta interfaz se muestra en la *figura 5.6*, la funcionalidad del sistema es de la siguiente manera:

- ***rs232rx***.La operación de este bloque consiste en recibir los datos de cada pixel de una imagen y enviarlos al bloque de control ***sram_ctrl.vhd***, así mismo, envía una señal al bloque ***fsm_sram.vhd*** para indicar que hay una dato a escribir, la señal **RDD** indica a este bloque que el dato ha sido procesado y que el sistema está listo para recibir el siguiente, dato esto se repite hasta el último dato recibido. Este bloque se encuentra en operación mientras la señal de entrada **R** se encuentre en '0'.

- ***fsm_sram***. Está máquina secuencial indica cuando se ha terminado de procesar un dato ya sea para escritura o lectura, y envía una señal al bloque ***sram_ctrl.vhd*** para indicar la operación que se está realizando. La señal H es enviada al bloque ***contador_m.vhd***.

- ***divisor_m.vhd***. Este bloque es un contador ascendente el cual nos da un retardo para la operación de lectura, el tiempo del retardo es determinado por **M**, cuando la señal de entrada **R** se encuentra en '1', el contador es activado y cuando el conteo llega a **M**, se envía la señal **Z** al bloque anterior para activar la operación de lectura.

- ***contador_m***. Este bloque nos lleva la cuenta de las direcciones de memoria para la operación de escritura.

- **sram_ctrl.** Este bloque lleva acabo todo el control para la interfaz con la memoria SDRAM, generando las siguientes señales:
  - **WE_ON**: esta señal prepara la memoria para la operación de escritura.
  - **OE_ON:** prepara la memoria para la operación de lectura.
  - **CE_A_N**: habilita la memoria para poder realizar cualquiera de las dos operaciones.
  - **FLASH_CE**: Habilita la memoria flash, en este proyecto no se utiliza, por lo tanto, siempre se encuentra en '1'.
  - **UB_A_N:** Byte más significativo de cada dato almacenado en alguna dirección
  - **LB_A_N**: Byte menos significativo.
  - **READY**: Indica cuando un dato fue almacenado o leído de alguna dirección de la memoria, sin embargo, para este trabajo esta señal no es empleada.
  - **Data_s2f_r:** envía los datos leídos de cada pixel que son leídos desde la memoria.
  - **Data_s2f_ur**: se utiliza como buffers de datos enviados de la memoria SDRAM al FPGA.

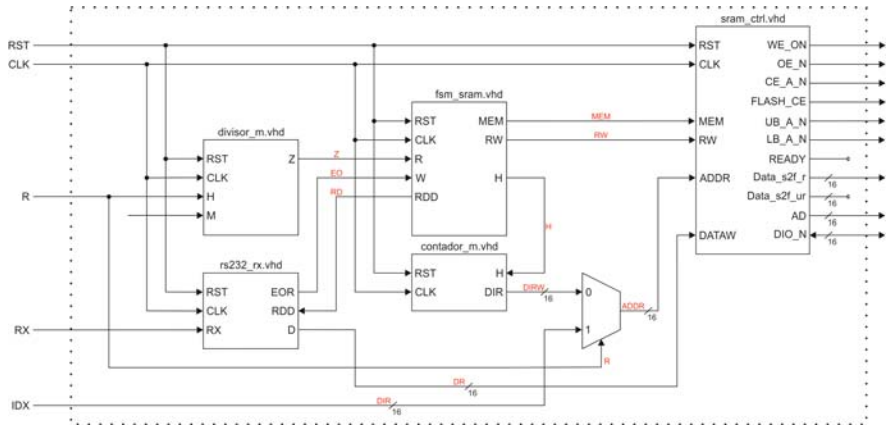


Figura 5.6. Diagrama de bloques de la Comunicación serial SDRAM mas a detalle

Agregando el bloque de la *figura 5.5* al sistema se obtiene el diagrama de la *figura 5.7*, donde agregamos la sincronía VGA y el generador de pixeles.

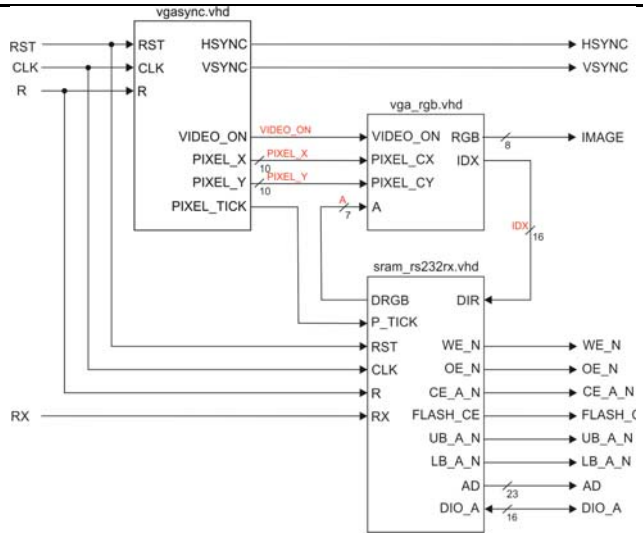


Figura 5.7. Diagrama de Bloques SDRAM

5.2. OPERACIONES DE TRANSFORMACIÓN DE LA IMAGEN

Para la aplicación de transformaciones de la imagen, se obtuvo el diagrama de la *figura 5*. Donde se implementan las siguientes transformaciones a dos imágenes:

1. **Escala de grises**

   Para esta operación lo que hicimos fue crear una LUT, en la cual tenemos los máximos y mínimos valores que puede obtener cada banda de color, los resultados se muestran en la *figura 6*. En la *tabla 1* se muestran los valores de cada banda de color.

Tabla 5.1. Valores para escala de grises

PROMEDIO RGB	R	G	B
0	**0**	**0**	**0**
1	**0**	**0**	**0**
2	**1**	**1**	**0**
3	**1**	**1**	**0**
4	**1**	**1**	**0**
5	**2**	**2**	**1**
6	**2**	**2**	**1**
7	**2**	**2**	**1**
8	**3**	**3**	**1**
9	**3**	**3**	**1**
10	**3**	**3**	**1**
11	**4**	**4**	**2**
12	**4**	**4**	**2**
13	**4**	**4**	**2**
14	**5**	**5**	**2**
15	**5**	**5**	**2**
16	**5**	**5**	**2**
17	**6**	**6**	**2**

2. Binarización

Para lograr la binarización de la imagen definimos un umbral, es decir un valor intermedio donde todos los valores del promedio RGB se encuentren por debajo o igual a él, se le asignará un valor de 0 y en caso de que sea mayor se asigna un valor de 1. Para este caso definimos el umbral igual a 8. Los resultados de esta operación se muestran en la *figura 7*.

3. Negativo de color

En esta operación lo que debemos realizar es una resta donde a nuestro valor máximo de cada banda de color le restamos el valor de cada pixel, para nuestro caso para R y G usaremos un valor de máximo de 7, y para B un valor máximo de 3. Los resultados obtenidos se muestran en la *figura 8*.

Al integrar las operaciones mencionadas anteriormente se obtiene un diagrama general del sistema que se muestra en la *figura 5.8*.

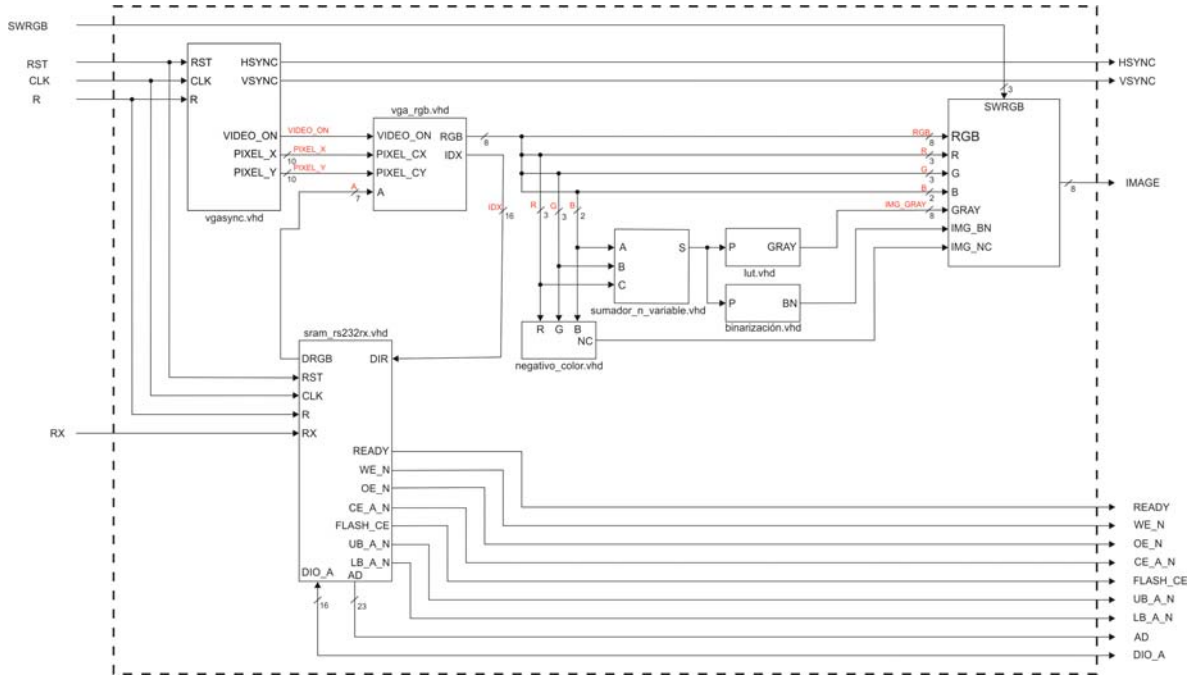


Figura 5.8. Sistema Básico de procesamiento de Imágenes

Las pruebas y resultados de esta implementación se observan en el capítulo 6.

El diagrama de la *figura 5.9*, queda como base para un trabajo a futuro el cual únicamente consiste en agregar los bloques de los filtros que se deseen implementar en el diagrama de la *figura 5.8*.

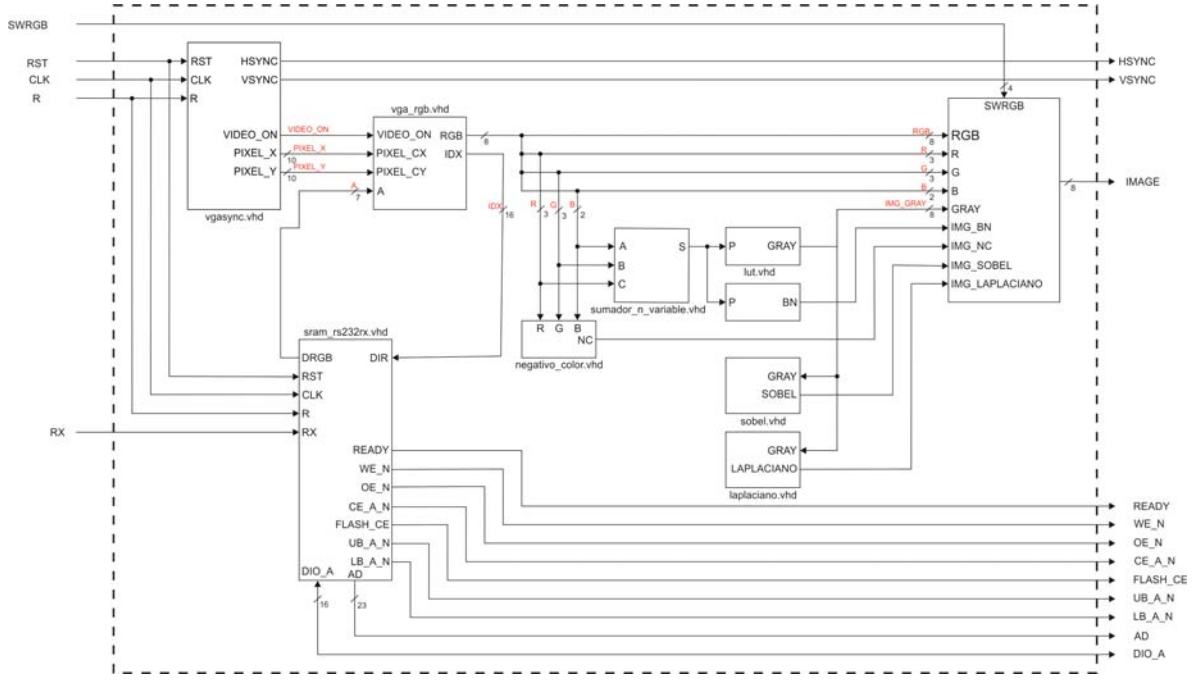


Figura 5.9. Sistema de procesamiento de imágenes con filtros

CAPÍTULO VI
PRUEBAS
Y
RESULTADOS

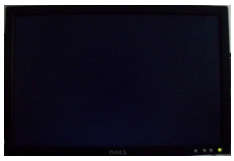
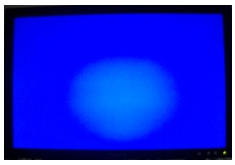
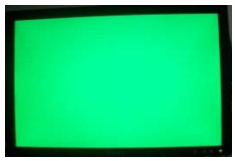
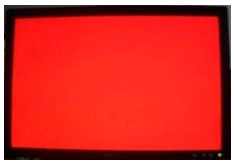
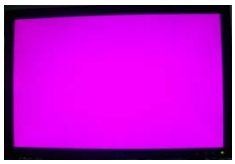
INTRODUCCIÓN

En este capítulo se tienen las pruebas realizadas y los resultados realizados en el proyecto de tesis. La explicación de cada prueba y resultado se describe en las siguientes secciones del capítulo.

6.1. GENERADOR RGB VGA

Los resultados obtenidos del circuito de sincronía del capítulo 3, se muestran en la *figura 6.1*. Cabe mencionar que la exhibición de los colores con cada tarjeta de desarrollo mostrando los 8 colores básicos es la misma, y las únicas adaptaciones que se hicieron para cada tarjeta fue el ajuste de la resolución del bus RGB.

Tabla 6.1. RGB_VGA

PRUEBAS DEL SISTEMA RGB_VGA			
 Negro	 Azul	 Verde	 Cyan
 Rojo	 Morado	 Amarillo	 Blanco

Esta prueba fue realizada con las tarjetas anteriormente mencionadas, con las cuáles se obtuvo el mismo resultado, sin embargo, las tarjetas Spartan 3AN y Nexys 2 tienen la capacidad de dar intensidades distintas de cada color. Los resultados comprueban las teorías de las tablas 3.1, 3.2 y 3.3. Lograda la sincronía se procede a realizar la prueba de la sección 6.2.

6.2. METODOLOGÍA PARA MANEJO DE IMÁGENES EN FPGA

En esta sección se tiene la prueba realizada con la metodología propuesta en la sección 5.1.1, donde, para lograr la exhibición de una imagen real previamente almacenada en el FPGA. Se utilizó una imagen, tal y como se visualiza en una PC o cualquier otro equipo que maneje los 24 bit en formato jpg. *La figura 6.1*(a) muestra la imagen a 24 bit, utilizada en el desarrollo de las pruebas, y de tamaño 256 x 256 pixeles. El sistema completo, con la pantalla y la tarjeta de desarrollo, se muestra en la figura *6.1*(b). La imagen es mostrada a partir de la posición (0,0), hasta la posición (255,255). La imagen resultante una vez aplicada la metodología, y almacenados los datos de la imagen en el FPGA, se muestra en la *figura 6.1*(c). Aquí se puede ver que existen marcadas diferencias en color, debido al ajuste de bits realizado. Sin embargo, aun así, se puede apreciar que el resultado es similar en forma a la *figura* original *6.1*(a).

a) b) c)

Figura 6.1. Resultado de la aplicación de la metodología. (a) Imagen original a 24 bit. (b) Sistema completo. (c) Imagen implementada en FPGA y ajustada a 8 bit.

Como resultado se obtiene una variación en el color comparada con la imagen original, por lo tanto, la imagen del FPGA no tiene una resolución en colores tan exacta como la original, esto debido a la normalización que fue necesaria hacer, como ya se mencionó en la sección 5.1.1. Sin embargo, al aplicar esta metodología, dio un resultado positivo obteniendo una base para leer imágenes en el FPGA y exhibirlas en el monitor, comprendiendo cómo usar los datos de una imagen en nuestro FPGA.

Cabe mencionar que este método lleva consigo ciertas desventajas si lo que se necesita es un proyecto de mayor complejidad como lo son:

1. Utiliza recursos del FPGA al 100%, por lo que en economía puede resultar de alto costo.

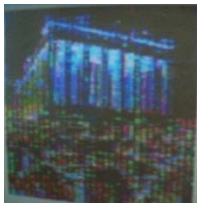
2. Generación del archivo .bit de hasta 4 horas, esto dependiendo de las características del equipo de trabajo (PC) donde es generado este archivo.

3. Debido al consumo de recursos, no es posible implementar algún filtro de procesamiento de imágenes en el mismo FPGA.

6.3. LECTURA DE IMÁGENES UTILIZANDO UNA MEMORIA SDRAM

En esta sección se muestran los resultados de la implementación de la sección 5.1.2, mostrados en la *tabla 6.2*, en donde, del lado izquierdo tenemos las imágenes originales (1 y 2) y las transformaciones aplicadas.

Tabla 6.2. Resultados de la Metodología 5.1.2

USANDO MEMORIA SDRAM			
Imagen en PC		Imagen en FPGA	
Imagen Original		Imagen Normalizada	
			
1	2	3	4
Negativo de una Imagen A Color			
			
5	6	7	8
Imagen en Escala de Grises			
			
9	10	11	12
Imagen Binarizada por Umbral			
			
13	14	15	16

Los resultados obtenidos, son parecidos a la metodología anterior, como ya se mencionó debido a la resolución de color en el puerto VGA.

Sin embargo, comparando las desventajas con la metodología anterior, esta metodología da ventajas sumamente importantes y significativas, las cuáles son:

1. Altas velocidades de cómputo. Esto debido a que los procesos o transformaciones a la imagen son llevadas a cabo de manera paralela.

2. Bajo Costo. Esto se refiere en cuanto a recursos del FPGA, ya que con la metodología anterior, se necesitó el 100% de ellos, con esta metodología y aplicando tres transformaciones a nuestra imagen, se obtuvo un consumo de recursos del 1%.

3. Debido a lo anterior, es posible, implementar aún más operaciones o filtros a este sistema, ya que se tiene disponible un 99% de los recursos del FPGA.

4. La generación del archivo .bit no supera los 2 minutos, y no se requiere de una PC con grandes características para una generación del mismo, con un tiempo aceptable.

La diferencia entre las imágenes obtenidas en una PC a las obtenidas en el FPGA de la tabla 6.2, es debido a la resolución de color. Sin embargo, a pesar de esta limitación, la imagen es bien apreciada, esto da una pauta para realizar un trabajo a futuro.

CONCLUSIONES

La ventaja de utilizar FPGA's, permite crear una arquitectura propia, que ayuda a mejorar el desempeño del sistema. Así mismo, debido a la diversidad de trabajo en el procesamiento de imágenes, dónde no será necesario utilizar una computadora convencional de propósito general, y al tener un hardware que tenga un propósito específico, da como resultado un ahorro de costos y se obtiene una eficiencia mayor por parte del sistema.

La metodología de la sección 5.1.1, dio pauta para trabajar con imágenes, utilizando dispositivos lógicos programables, como el FPGA, siendo una base para la generación de trabajos futuros. Utilizando el lenguaje descriptivo de hardware VHDL, respetando el estándar del IEEE, para no perder portabilidad a cualquier tecnología de FPGA's disponible. Esta metodología fue aceptada como artículo en un congreso, así mismo fue presentado como ponencia en abril de 2010. El artículo se encuentra en la sección de anexos.

En la metodología de la sección 5.1.2 se propone una manera de realizar el procesamiento de imágenes en hardware, teniendo altas velocidades de cómputo, y a un bajo costo, ya que el consumo de recursos hasta este punto permite realizar un trabajo a futuro más complejo.

Por lo tanto, como trabajo a futuro se propone, la implementación de filtros con operaciones más complejas, modificar o desarrollar una tarjeta con puerto VGA de tal manera que se aumente la resolución de video para mejores resultados. De la misma manera, se tiene contemplado trabajar en la adquisición de imágenes en tiempo real, así como el procesamiento de las mismas, utilizando las ventajas que nos proporciona. Para esto, es necesario el uso de memorias dinámicas, para almacenar la imagen, y poder aplicar procesamiento de imágenes, como filtros, y obtener resultados de una manera más rápida y dinámica.

ANEXOS

TEMATICA: Procesamiento de Imágenes

Metodología para manejo de imágenes en FPGA
Methodology for Image Handling on FPGA

Carlos Alberto Ramos Arreguín, Orlando Marcos Cora Gallardo, Juan Manuel Ramos Arreguín, Jesús Carlos Pedraza Ortega, Sandra Luz Canchola Magdaleno, José Emilio Vargas Soto
CIDIT Facultad de Informática de la Universidad Autónoma de Querétaro.

RESUMEN. En este trabajo se propone una metodología estándar para manejo de imágenes en formato JPG, utilizando técnicas de lenguaje descriptivo en hardware, para su implementación en FPGA, incluyendo el sistema de sincronía de la imagen y el barrido de la pantalla. Debido a los recursos de la tarjeta empleada, es necesario ajustar la resolución de video de 14 bit a 8 bit. Los resultados se presentan, comparando la imagen resultante de 8 bit, con la imagen original de 24 bit.

Palabras Clave. Imágenes JPG, FPGA, lenguaje descriptivo de hardware.

1. INTRODUCCIÓN

El creciente desarrollo de las tecnologías y, en especial de la información, hace que cada día sea más frecuente el desarrollo de aplicaciones para el tratamiento digital de imágenes. Por otra parte, el desarrollo de equipos cada vez más sofisticados en diversos campos del procesamiento digital de imágenes, han justificado el desarrollo tecnológico alrededor del tratamiento digital de imágenes. A modo de ejemplo, encontramos dos casos de especial relevancia:

a) *Medicina,* donde los sofisticados equipos PET (***Positrón Emisión Tomography***), resonancia magnética, rayos X, etc., tienen en el tratamiento digital de imágenes un alto grado de importancia en los diagnósticos, los cuales no podrían ser posibles, sin este tipo de procesos.

b) *Observación de la Tierra,* donde los sensores acoplados en los satélites artificiales son capaces de proporcionar imágenes en las que se pueden observar detalles de hasta 0.6 metros de tamaño. Gracias al tratamiento de las imágenes que generan, es posible detectar zonas deforestadas, evolución de fenómenos meteorológicos, etc.

Hoy en día el uso de dispositivos lógicos programables, en particular los FPGA's (Field Programmable Gate Arrays) (Toledo Ana, 2005), ha crecido de una manera muy significativa en el área de software embebido, debido a que son tan eficientes como un ASIC (Application Specific Integrated Circuit) y compiten en costo con los microcontroladores (Quintero, 2006). Cabe mencionar que estos dispositivos pueden procesar una gran cantidad de información en muy poco tiempo, a comparación de una PC convencional. En este artículo se presenta una metodología para el manejo de imágenes en un dispositivo FPGA. En este caso, no se está haciendo ningún proceso de imágenes, y se enfoca a la manera como se puede generar el código en VHDL, utilizando únicamente la librería del IEEE, en forma de tabla de búsqueda (Look Up Table, LUT), y la manera como se muestra la imagen en pantalla, lo cual es importante cuando se considera el desarrollo de aplicaciones que requieren el procesamiento digital de imágenes en tiempo real.

En la actualidad, se almacenan imágenes en FPGA en escala de grises, esto si se desea aplicar inmediatamente un filtro a la imagen, o por medio de una herramienta no estándar, lo cual es posible mediante el software MemUtil de Digilent, con el cual se almacena la información de una imagen en una memoria de tipo SDRAM(Procesado de Imagen en FPGA, 2009). Así mismo hay aplicaciones ya desarrolladas, las cuáles no utilizan únicamente el estándar de la IEEE, sino que también se basan en librerías del fabricante. Se utilizó una tarjeta de desarrollo NEXYS 2 de Digilient, con un circuito SPARTAN 3E, de Xilinx, con número de parte XC3S1200E.

A continuación se menciona la manera cómo se desarrolla la implementación de una imagen en hardware.

DESARROLLO

2.1 Representando Imágenes Digitales

Una imagen no es otra cosa que una matriz de datos, donde el tamaño de la matriz depende directamente del tamaño de la imagen, y cada dato representa el valor del pixel. El valor del pixel representa la información del color, que puede ser del tipo RGB, por ejemplo. Las imágenes cuando son adquiridas, representan una muestra de la vida cotidiana, por lo que podemos hablar de una imagen muestreada. El resultado del muestreo de una imagen, es una matriz de valores enteros de tamaño MxN, donde se ha cuantificado el valor de cada pixel. Asumiendo que una imagen es una función $f(r,c)$, que es muestreada y como resultado de la imagen digital se tiene N renglones y M columnas. Los renglones están representados por r y las columnas por y. Los valores de los pixeles son almacenados en la matriz, por renglones, de izquierda a derecha, hasta completar el total de pixeles de la imagen en cuestión. M representa el número de pixeles horizontales y N el número de pixeles verticales. Un ejemplo puede ser una resolución de 256x256. Esto se ilustra en la figura 1.

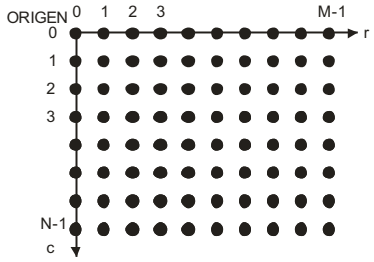


Figura 1. Coordenadas de los pixeles

En la actualidad, para realizar procesamiento de imágenes se utiliza una computadora convencional de propósito general, utilizando por ejemplo, MATLAB, para llevar a cabo dicha actividad. En el proceso, primero se adquiere la imagen (video, cámara digital, escáner, etc.), enseguida en MATLAB se aplica algún filtro, y por último la imagen es procesada y mostrada en el monitor, como se muestra en la figura 2.

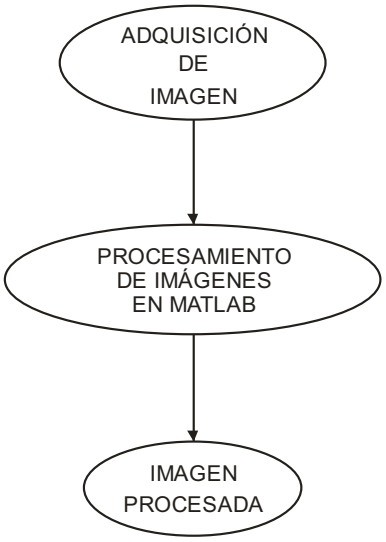


Figura 2. Procesamiento de Imágenes Convencional

Este proceso es implementado en un sistema embebido, basado en FPGA, debido a las ventajas mencionadas previamente. Sin embargo, para realizar esto, es necesario realizar un proceso con la imagen, dado que en el sistema digital en el FPGA la imagen debe ser introducida en valor digital, donde cada valor representa el color de un punto del monitor (pixel). En la figura 3, se muestra el proceso general de la manera como debe realizarse el proceso de representación de una imagen en FPGA.

En el proceso de la figura 3, la imagen que estamos usando tiene un formato jpg, que puede ser adquirida por cualquier método convencional. Una vez que se cuenta con la imagen, se utiliza el MATLAB para realizar la conversión de la imagen a un archivo descriptivo de hardware, que en este caso se utiliza VHDL, con lo que se va a contar con un módulo, el cual simplemente debe ser agregado a un módulo de mayor jerarquía. En seguida, se pasa al bloque de sincronía, donde se sincroniza los datos de la imagen con el barrido de la pantalla, para su correcta visualización. Más adelante se habla más a detalle del controlador VGA, que es otro módulo incluido en el mismo archivo descriptivo que incluye la descripción de la imagen. Finalmente, tenemos el bloque de la pantalla, que es donde se visualiza la imagen.

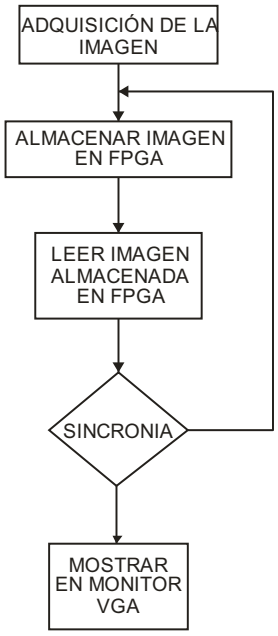


Figura 3. Procesamiento de Imágenes en Hardware

El proceso de la figura 3, se compone de diversas etapas, por lo que es necesario explicar un poco más cada uno de los bloques mostrados en los siguientes puntos.

2.2 Imagen.

Es importante mencionar que los archivos para PC de una imagen de formato jpg, manejan el formato de colores tridimensional RGB (Rojo, Verde, Azul), lo que significa que, para representar el color de un pixel, se utilizan 24 bit, es decir, 8 bit para cada color. Sin embargo, en el caso de la tarjeta a utilizar, se manejan solamente 8 bit de video, donde 3 son para rojo, 3 para verde y 2 para azul. Debido a esto, es necesario realizar un ajuste de bit para poder realizar un manejo adecuado de la imagen, lo cual se realiza en el siguiente bloque del proceso.

En el caso de imágenes a color, los elementos de la matriz, vienen dados por tres valores, que representan cada uno de los componentes básicos del color en cuestión. Estos componentes son el *Rojo (R), Verde (G) y Azul (B),* el conocido código *RGB*, en el caso general de video, el conjunto de valores (0,0,0) es el negro absoluto; el (255,255,255) el blanco absoluto; el (255, 0, 0) el rojo puro; (0,255,0) el verde puro; y el (0,0,255) el azul puro. Sin embargo, debido a que la tarjeta a utilizar maneja solamente 8 bit, el negro absoluto en esta caso es (0,0,0); el blanco es (7,7,3); el rojo (7,0,0); el verde (0,7,0) y el azul (0,0,3). Como es lógico, la combinación de distintos valores proporciona otros colores, pero con 8 bit se tiene una combinación menor que con 24 bit. Por lo tanto, una imagen en color posee tres bandas espectrales, rojo, verde y azul; cada una de ellas es una matriz de números 2D con valores en el rango 0 a 255 para imágenes de 8 bit. A veces para representar una imagen digital a color, se utiliza una representación basada en una matriz de valores y un mapa de colores o paleta asociada.

El puerto VGA de la tarjeta NEXYS 2, tiene cinco señales activas, incluyendo la sincronía horizontal y vertical (HSYNC, VSYNC), y tres señales de video: rojo, verde y azul (RGB), y se encuentran físicamente dirigidas al conector de video tipo D de 15 pines. La señal de video es analógica y el controlador de video utiliza un convertidor digital a analógico, para convertir la salida digital al nivel analógico deseado. En esta tarjeta, las señales de video RGB pueden generar solamente 2^8 colores diferentes, como se mencionó en el punto 2.2.

2.3 Representación en VHDL.

Debido a lo explicado en los puntos 2.1 y 2.2, la imagen de formato jpg debe ser representado en un archivo descriptivo de hardware, que en este caso, utilizamos VHDL para tal propósito. En la figura 4, se muestra un diagrama de flujo del proceso a seguir para realizar la conversión de la imagen a un archivo descriptivo de hardware.

Para la implementación del proceso de la figura 4, se utiliza MATLAB, por las facilidades que presenta en el manejo de las imágenes. El primer paso, es asignar la imagen a una matriz M. la matriz M generada, es de 3 dimensiones, por lo que, en el segundo bloque, el valor obtenido para *j* es el número de renglones, para *k* el número de columnas, y para *l* es de 3, pues la matriz es tridimensional.

En seguida, es necesario hacer el ajuste de los valores de la imagen M de 24 bit, para poder obtener una imagen N de 8 bit. Para esto, se hace uso de dos ciclos anidados con las variables *r* (número de renglón), y *c* (número de columnas). Cada valor RGB de la matriz M debe ser dividido entre 255, para normalizar el valor del pixel, y posteriormente se debe multiplicar por 7, para ajustar a 3 bit, y por 3 para ajustar a dos bit. Tales ecuaciones se muestran en la figura 4. Finalmente, el último bloque representa la generación de una tabla de acceso a datos (LUT), que también, en cierta forma, se puede interpretar como una memoria ROM interna en el FPGA.

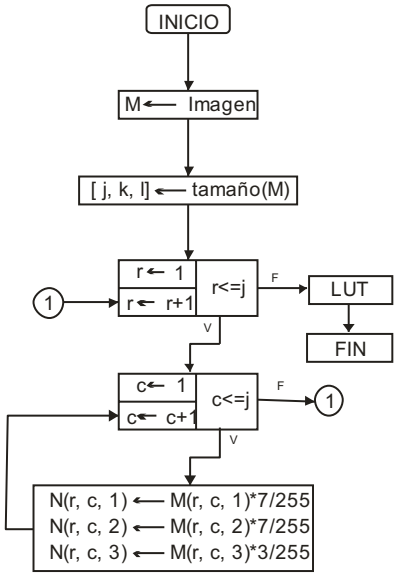


Figura 4. Diagrama de flujo del proceso de conversión de una imagen a lenguaje descriptivo de hardware.

Para generar el archivo descriptivo de hardware, se deben seguir los siguientes pasos.

1. Crear un archivo texto con el nombre de la imagen, con extensión *vhd*.
2. Se graba al archivo el encabezado de la descripción.
3. Determinar el número de bit para la dirección de datos, de acuerdo al tamaño de la matriz. Por lo pronto, nos referiremos al número de bit como *n*. se está considerando una imagen donde el número de renglones es igual al número de columnas, y el tamaño es potencia de dos.
4. La dirección de entrada a la LUt se forma con el número de renglón en la parte más significativa y el número de columna en la parte menos significativa.
5. Se envía al archivo la entidad, incluyendo los puertos de dirección, y los datos de salida para el video.
6. Se genera la sección de arquitectura, con su begin.
7. Se genera el proceso, donde la dirección es utilizada en un proceso *case*.

8. Se convierte el valor de la dirección a una cadena que contiene el valor en binario, así como el dato de salida, que debe convertirse igualmente a binario, en otra cadena.
9. Se genera cada uno de los casos utilizando las cadenas con las conversiones a binario, y se almacenan en el archivo texto, cuidando la sintaxis del lenguaje descriptivo de hardware.
10. Se cierra el proceso.
11. Se cierra la arquitectura.
12. Se agrega la descripción a un módulo de mayor jerarquía.

Una vez realizado lo anterior, ya puede presentar en un monitor la imagen ajustada a 8 bit. A continuación, se describe la manera como se maneja la sincronía de la imagen con la pantalla.

2.4 Sincronía

Es necesario mencionar que, para mostrar una imagen en la pantalla, se tiene un haz de electrones que, al impactar en la pantalla, generan el color, en cualquier monitor que soporte VGA. La figura 5, muestra el barrido del haz de electrones en una pantalla. Debido a esto, es necesario sincronizar el barrido con la generación de la imagen, para que se pueda tener una correcta visualización. (Pong, 2008).

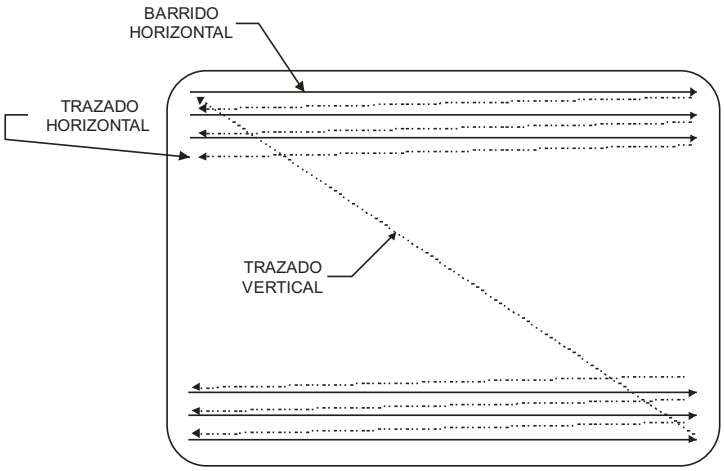


Figura 5. Secuencia de barrido del haz de electrones para mostrar una imagen en pantalla.

Por lo tanto, es necesario un controlador de video para generar las señales de sincronización entre el barrido de los pixeles y los datos del color de cada pixel. En la figura 6 se muestra un diagrama simplificado de un controlador VGA. Contiene un circuito de sincronización, etiquetado VGA-SYNC, y un circuito de generación de pixel.

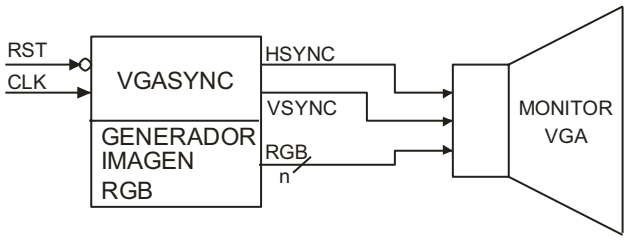


Figura 6. Diagrama de Bloques

El circuito VGA-SYNC, genera el tiempo y las señales de sincronización, las señales HSYNC y VSYNC están conectadas al puerto VGA para controlar el barrido horizontal y vertical del monitor. Las dos señales son decodificadas desde los contadores internos las cuales son PIXEL_X y PIXEL_Y, que indican la posición del barrido y esencialmente la localización del pixel actual. El circuito también genera la señal VIDEO_ON, que indica cuando habilitar o deshabilitar el monitor.

El generador imagen RGB, se encarga de generar el color de cada pixel, de acuerdo al valor de la posición actual del pixel a mostrar, lo cual se forma con las coordenadas de las señales (PIXEL_X, PIXEL_Y), así como con el control externo.

3. PRUEBAS Y RESULTADOS

La metodología propuesta fue probada utilizando una imagen, tal y como se visualiza en una PC o cualquier otro equipo que maneje los 24 bit del formato jpg. La figura 7a muestra la imagen a 24 bit, utilizada en el desarrollo de las pruebas, y de tamaño 256 x 256 pixeles. El sistema completo, con la pantalla y la tarjeta de desarrollo NEXYS 2, se muestran en la figura 7b. La imagen es mostrada a partir de la posición (0,0), hasta la posición (255,255). La imagen resultante una vez aplicada la metodología, y almacenada la figura en el FPGA, se muestra en la figura 7c. Aquí se puede ver que existen marcadas diferencias en color, debido al ajuste de bits realizado. Sin embargo, aún así, se puede apreciar que el resultado es similar en forma a la figura original 7a.

(a) (b) (c)

Figura 7: Resultado de la aplicación de la metodología.
(a) Imagen original a 24 bit.
(b) Sistema completo.
(c) Imagen implementada en FPGA y ajustada a 8 bit.

CONCLUSIONES

Esta metodología establece una manera como podremos trabajar el manejo de imágenes, utilizando dispositivos lógicos programables, como el FPGA, siendo una base para la generación de trabajos futuros. Se está utilizando el lenguaje descriptivo de hardware VHDL, respetando el estándar del IEEE, para no perder portabilidad a cualquier tecnología de FPGA's disponible.

La ventaja de utilizar FPGA's, permite crear una arquitectura propia, que nos permita mejorar el desempeño del sistema. Así mismo, debido a la diversidad de trabajo en el procesamiento de imágenes, dónde no será necesario utilizar una computadora convencional de propósito general, y al tener un hardware que tenga un propósito específico, nos ayuda en ahorro de costos y podremos obtener una eficiencia mayor por parte del sistema.

Como parte de trabajos a futuro, está el manejo de memorias dinámicas, para almacenar la imagen, y poder aplicar procesamiento de imágenes, como filtros, y obtener resultados de una manera más rápida y dinámica.

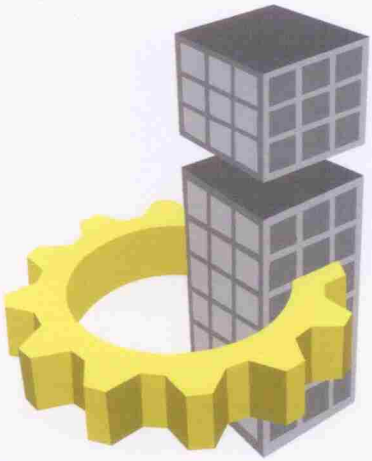
4. REFERENCIAS

1. Toledo Ana, Vicente-Chicote Cristina, Suardíaz Juan, Cuenca Sergio;Xilinx System Generator Based HW Components for Rapid Prototyping of Computer Vision SW/HW Systems; Springer Verlag; pp. 667-674; 2005.

2. Quintero M. Alexander, Vallejo R. Eric; Image Processing Algorithms using FPGA; Revista Colombiana de Tecnologías de Avanzada; Vol. 1; No. 7; 2006; pags 11 a 16; ISSN: 1692-7257.

3. http://procesadodeimagenenfpga.wordpress.com/

4. Pong P. Chu; FPGA Prototyping by VHDL Examples Xilinx Spartan – 3 Version, Wiley Interscience; New Jersey, 1ª Edición, USA, 2008.



6° CONGRESO INTERNACIONAL DE INGENIERÍA

La UNIVERSIDAD AUTÓNOMA DE QUERÉTARO
a través de la FACULTAD DE INGENIERÍA,
otorga el presente

RECONOCIMIENTO

a:

CARLOS ALBERTO RAMOS ARREGUÍN

Por haber participado con la ponencia:

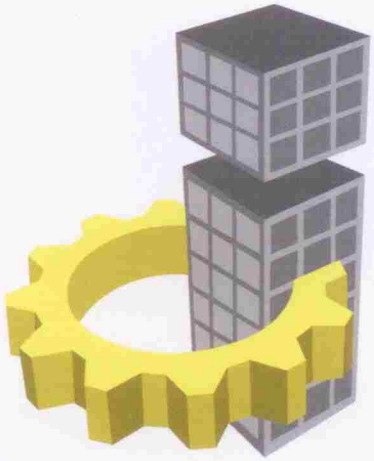
"METODOLOGÍA PARA MANEJO DE IMÁGENES EN FPGAS"

en el Congreso Internacional de Ingeniería en su sexta edición,
realizado del 21 al 23 de Abril del 2010 en
la ciudad de Santiago de Querétaro, México.

CONCYTEQ

Dr. Gilberto Herrera Ruiz
Director Facultad de Ingeniería

M.L. Adriana Medellín Gómez
Comité Organizador



6° CONGRESO INTERNACIONAL DE INGENIERÍA

La UNIVERSIDAD AUTÓNOMA DE QUERÉTARO
a través de la FACULTAD DE INGENIERÍA,
otorga el presente

RECONOCIMIENTO

a:

CARLOS ALBERTO RAMOS ARREGUÍN

Por haber participado con la ponencia:

"METODOLOGÍA PARA MANEJO DE IMÁGENES EN FPGAS"

en el Congreso Internacional de Ingeniería en su sexta edición,
realizado del 21 al 23 de Abril del 2010 en
la ciudad de Santiago de Querétaro, México.

CONCYTEQ

Dr. Gilberto Herrera Ruiz
Director Facultad de Ingeniería

M.L. Adriana Medellín Gómez
Comité Organizador

REFERENCIAS BIBLIOGRÁFICAS

1. Pajares Gonzalo, Sanz Martin; Visión por Computador, Imágenes Digitales y Aplicaciones; Alfaomega, 2ª Edición; 2008; ISBN: 978-84-7897-831-1.

2. Moya J., González Z, Pedraza J., Soto A., Delgado M., Canchola S., Caracterización de un Sistema de Reconstrucción de Objetos 3D, 6º Congreso Internacional de Ingeniería, Querétaro, Qro., México; 2010.

3. Quintanar-Pérez M.E., Moya-Morales J.C., Pedraza-Ortega J.C., Canchola-Magdaleno S.L., Gorrostieta-Hurtado E. senior member, IEEE, Aceves-Fernández M.A., Ramos-Arreguín J.M.; Propuesta de Reconstrucción 3D utilizando Transformada Wavelet; VII Congreso Internacional en Innovación y Desarrollo Tecnológico, CIINDET 2009; pag. 1 a pag. 5; ISBN: 978-607-95255-1-4.

4. Pedraza Ortega Jesús Carlos, Rodríguez Moreno José Wilfrido, Barriga Rodríguez Leonardo, Gorrostieta Hurtado Efrén, Salgado Jiménez Tomás, Ramos Arreguín Juan Manuel, Rivas Ángel; Image Processing for 3D Reconstruction Using a Modified Fourier Transform Profilometry Method; MICAI 2007, Edit. Springer; pag. 705 a 712; ISSN: 0302-9743.

5. Pedraza Ortega J. C., Canchola Magdaleno S. L., Gorrostieta Hurtado E., Aceves Fernández M. A., Ramos Arreguín J. M., Delgado Rosas M.; Three Dimensional Reconstrucción System base don a Segmentation Algorithms and a Modified Fourier Transform Profilometry; CERMA 2009; págs. 344 a 348; ISBN: 978-0-7695-3799-3

6. Tusch Michael; High-Performance Image Processing on FPGAs; Xcell Journal, Xilinx; pages 42 to 44; 2006.

7. Toledo Ana, Vicente-Chicote Cristina, Suardíaz Juan, Cuenca Sergio; Xilinx System Generator Based HW Components for Rapid Prototyping of Computer Vision SW/HW Systems; Springer Verlag; pp. 667-674; 2005.

8. Sánchez Martínez Miguel Ángel, Diseño en FPGA de un Circuito Comparador de imágenes. Tesis de Maestría, Centro de investigación y de Estudios Avanzados del Instituto Politécnico Nacional, Depto. De Ingeniería Eléctrica, Sección Computación; México; Julio 2005.

9. Quintero M. Alexander, Vallejo R. Eric; Image Processing Algorithms using FPGA; Revista Colombiana de Tecnologías de Avanzada; Vol. 1; No. 7; 2006; págs. 11 a 16; ISSN: 1692-7257.

10. Bravo Muñoz Ignacio; Arquitectura Basada en FPGA para la Detección de Objetos en movimiento, utilizando Visión Computacional y Técnicas PCA. Tesis de Doctorado, Departamento de Electrónica de la Escuela Politécnica de la Universidad de Alcalá; España, 2007.

11. Romero Troncoso René de Jesús; Electrónica Digital y Lógica Programable; Universidad de Guanajuato, 2ª Edición; Guanajuato, México, 2007; ISBN: 968-864-449-8.

12. Galeano Gustavo, Programación de Sistemas Embebidos en C, Alfaomega, 1ª Edición; México DF, 2009; ISBN: 978-958-682-770-6

13. Rahul Dubey, Introduction to Embedded System Design using Field Programmable Gate Arrays, Springer-Verlag London Limited, India, 2009, ISBN 978-1-84882-015-9.

14. Walls Colin, Embedded Software: The Works, Elsevier, USA, 2006, ISBN 0-7506-7954-9.

15. Ganssle Jack, Noerggard Tammy, Eady Fred, Edwards Lewin, Katz David J., Gentile Rick, Arnold Ken, Hyder Kamal, Perrin Bob, Huddleston Creed; Embedded Hardware; Springer, 2008; ISBN 978-0-7506-8584-9.

16. Branislav Kisacanin, Shuvra S. Bhattacharyya, Embedded Computer Vision, Springer; Londrés, 2009; ISSN 1617-7916, ISBN 978-1-84800-303-3.

17. Pong P. Chu; FPGA Prototyping by VHDL Examples Xilinx Spartan – 3 Version, Wiley Interscience; New Jersey, 1ª Edición, USA, 2008; ISBN 978-0-470-18531-5.

18. Gonzalez Rafael C., Woods Richard E.; Digital Image Processing; Prentice Hall, 3ª Edición; USA, 2008; ISBN: 0-201-18075-9.

19. Ramos Arreguín Carlos Alberto, Cora Gallardo Orlando Marcos, Ramos Arreguín Juan Manuel, Pedraza Ortega Jesús Carlos, Canchola Magdaleno Sandra Luz, Vargas Soto; Metodología para Manejo de Imágenes en FPGA; 6º Congreso Internacional de Ingeniería, Querétaro, Qro. México, 2010; ISBN: 978-607-7740-39-1.