

**INSTITUTO TECNOLÓGICO DE CHIHUAHUA**  
**DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN**

---

***“PLATAFORMA SoC FPGA PARA ANÁLISIS DE  
ALGORITMOS DE PROCESAMIENTO DIGITAL  
DE IMÁGENES EN TIEMPO REAL”***

***TESIS***

**QUE PARA OBTENER EL GRADO DE**

***MAESTRO EN CIENCIAS  
EN INGENIERÍA ELECTRÓNICA***

**PRESENTA:**

***LUIS GERARDO ESQUIVEL RÍOS***

**DIRECTOR DE LA TESIS:  
DR. JAVIER VEGA PINEDA**

**CHIHUAHUA, CHIH., AGOSTO 2020**



**EDUCACIÓN**  
SECRETARÍA DE EDUCACIÓN PÚBLICA



TECNOLÓGICO  
NACIONAL DE MÉXICO





Chihuahua, Chih. 18 de agosto de 2020

**C. LUIS GERARDO ESQUIVEL RÍOS  
PRESENTE**

Por este conducto le comunico que a propuesta del Jurado de Examen, la División de Estudios de Posgrado e Investigación ha concedido autorización para la impresión de su tesis para obtener el grado de Maestro en Ciencias en Ingeniería Electrónica, cuyo título es:

***"Plataforma SoC FPGA para análisis de algoritmos de procesamiento digital de imágenes en tiempo real"***

La tesis presenta el siguiente contenido de capítulos:

- I Introducción
- II FPGA e integración de sistemas de un solo chip (SoC) Zynq y hardware
- III Algoritmos para procesamiento digital de imágenes
- IV Desarrollo del sistema en SoC FPGA
- V Vivado high level synthesis (HLS)
- VI Diseño de módulos en vivado HLS para procesamiento de imágenes
- VII Implementación del sistema y resultados
- VIII Conclusiones

**ATENTAMENTE**

Excelencia en Educación Tecnológica®  
"La Técnica por el Engrandecimiento de México"

**MTRO. ROGELIO ENRIQUE BARAY ARANA  
JEFE DE LA DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN**





Chihuahua, Chih. 18 de agosto de 2020

**MTRO. ROGELIO ENRIQUE BARAY ARANA**  
**JEFE DE LA DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN**  
**PRESENTE**

Por medio de la presente notificamos a usted que en cumplimiento de los requerimientos para la obtención del grado de Maestro en Ciencias en Ingeniería Electrónica, el documento de tesis del **C. LUIS GERARDO ESQUIVEL RÍOS**, ha sido aprobado y aceptado para su impresión. El título de la tesis es:

***"Plataforma SoC FPGA para análisis de algoritmos de procesamiento digital de imágenes en tiempo real"***

Por lo que proponemos, le sea concedida la autorización de impresión correspondiente.

Agradeciendo la atención a la presente, quedamos de usted:

**ATENTAMENTE**

*Excelencia en Educación Tecnológica»  
"La Técnica por el Engranaje de México"*



**DR. JAVIER VEGA PINEDA**  
**DIRECTOR DE TESIS**

**DR. LUIS FRANCISCO CORRAL MARTÍNEZ**  
**MIEMBRO DEL JURADO DE EXAMEN**

**DR. JOSÉ LUIS DURÁN GÓMEZ**  
**MIEMBRO DEL JURADO DE EXAMEN**

**DR. JOSÉ RIVERA MEJÍA**  
**MIEMBRO DEL JURADO DE EXAMEN**





# EDUCACIÓN

SECRETARÍA DE EDUCACIÓN PÚBLICA



TECNOLÓGICO  
NACIONAL DE MÉXICO

Instituto Tecnológico de Chihuahua

"2020, Año de Leona Vicario, Benemérita Madre de la Patria"

## CARTA CESIÓN DE DERECHOS

En la ciudad de Chihuahua el día 18 de agosto de 2020, el que suscribe C. LUIS GERARDO ESQUIVEL RÍOS de la Maestría en Ciencias en Ingeniería Electrónica, con número de control G18060026, adscrito a la División de Estudios de Posgrado e Investigación del Instituto Tecnológico de Chihuahua, manifiesta que es autor intelectual del presente trabajo de Tesis bajo la dirección del Dr. Javier Vega Pineda y cede los derechos del trabajo titulado "Plataforma SoC FPGA para análisis de algoritmos de procesamiento digital de imágenes en tiempo real", al Tecnológico Nacional de México y/o Instituto Tecnológico de Chihuahua para su difusión, divulgación, transmisión, reproducción, así como su digitalización con fines académicos y de investigación.

**C. LUIS GERARDO ESQUIVEL RÍOS**





## DECLARACIÓN DE ORIGINALIDAD

En la ciudad de Chihuahua el día 18 de agosto de 2020, el que suscribe C. LUIS GERARDO ESQUIVEL RÍOS de la Maestría en Ciencias en Ingeniería Electrónica, con número de control G18060026, adscrito a la División de Estudios de Posgrado e Investigación del Instituto Tecnológico de Chihuahua, manifiesta que es autor intelectual de la tesis titulada "Plataforma SoC FPGA para análisis de algoritmos de procesamiento digital de imágenes en tiempo real" bajo la dirección del Dr. Javier Vega Pineda; que el contenido es original y que las fuentes de información consultadas para su fundamentación están debidamente citadas y referenciadas.

**C. LUIS GERARDO ESQUIVEL RÍOS**



Chihuahua, Chihuahua, a 18 de agosto del año 2020

Dra. María Elena Álvarez-Buylla Roces  
Directora del CONACYT

Por este conducto aprovecho la ocasión para saludarla e informarle que a la fecha he obtenido el grado de **Maestría en Ciencias de Ingeniería Electrónica** en la División de Estudios de Posgrado e Investigación del Instituto Tecnológico de Chihuahua. Agradezco todo el apoyo brindado por CONACYT ya que el otorgamiento de la beca para estudios de Posgrado, permitió dedicarme de tiempo completo al programa y con ello lograr el cumplimiento del objetivo principal del convenio establecido.

Sin otro particular por el momento, me es grato quedar de usted como su seguro servidor, no sin antes reiterar nuevamente todo mi agradecimiento.

Atentamente

A handwritten signature in blue ink, reading "Luis Gerardo Esquivel Ríos", is written over a horizontal line.

Luis Gerardo Esquivel Ríos  
Exbecario CONACYT

---

## AGRADECIMIENTOS

Agradezco a mis padres Manuel Esquivel y María Ríos, por haberme brindado todo el apoyo a lo largo de mi formación académica. De igual manera agradezco a mis hermanos y sobrinos quienes siempre estuvieron pendientes de mi dando apoyo y cariño. Doy gracias a toda mi familia por forjarme como persona mediante enseñanzas que lograron crearme como persona y gracias a ellos no fuera posible este logro que es dedicado a ellos.

Al Dr. Javier Vega Pineda por darme la oportunidad de trabajar bajo su instrucción, obteniendo grandes conocimientos académicos como personales gracias a sus asesorías y consejos en el trascurso de mi estancia en el Instituto Tecnológico de Chihuahua. No me queda más que decir gracias por tratarme como persona y como su alumno todo este tiempo.

Gracias a cada uno de mis profesores por brindarme los conocimientos necesarios para dar por concluida mi investigación. En especial al Dr. Luis Francisco Corral y Dr. José Luis Durán por instruirme en cada uno de los comités y darse su tiempo para revisar esta tesis y darme retroalimentación para obtener un trabajo de calidad.

Un especial agradecimiento a la Lic. Francisco Javier Cruz Baca por su apoyo a lo largo de mi formación académica, por sus consejos. Reitero mi respeto y admiración hacia su trabajo, persona y gracias por más que profesor ser un gran amigo. También agradecer al Lic. Marco Antonio Ozuna González quien es instructor de la Banda de Guerra y Escolta del ITCH y a lo largo de mi estancia en la institución siempre recibí su apoyo completamente siendo un grandioso profesor que me brindo su amistad.

Al personal de posgrado, por estar al pendiente de mi formación y en especial a la M.C. Alma Corral.

Gracias a mis compañeros y amigos de los laboratorios, con quienes compartí aventuras personales, conocimientos y más que todo eso nos tratamos como familia. No me queda más que decir gracias por su amistad y compañerismo siempre.

**RESUMEN****PLATAFORMA SoC FPGA PARA ANÁLISIS DE ALGORITMOS DE PROCESAMIENTO DIGITAL DE IMÁGENES EN TIEMPO REAL**

Ing. Luis Gerardo Esquivel Ríos

Maestría en Ciencias en Ingeniería Electrónica

División de Estudios de Posgrado e Investigación del

Tecnológico Nacional de México - Instituto Tecnológico de Chihuahua

Chihuahua, Chih. 2020

Director de Tesis: Dr. Javier Vega Pineda

Se presenta un sistema para la experimentación y evaluación de algoritmos orientados al procesamiento digital de imágenes implementado en una plataforma de desarrollo cuyo núcleo es un dispositivo *Zynq* de Xilinx con características duales de hardware (FPGA - microcontrolador ARM Cortex-A9).

El procesamiento de imágenes y video en ambientes digitales es conocido por el consumo grande de recursos computacionales, tanto en almacenamiento de memoria como en tiempo de procesamiento. Es por tanto un reto el lograr contar elementos alternativos a los tradicionales para implementar sistemas que permitan el óptimo desempeño de los algoritmos que trabajan sobre datos de imágenes y video.

El sistema desarrollado cuenta con cámara de video y memoria para la adquisición y almacenamiento de cuadros o “*frames*”, un esquema híbrido de procesamiento de los cuadros (conjunto de píxeles) e interfaz de salida VGA para el despliegue de la imagen procesada. El esquema híbrido de procesamiento, núcleo del sistema es un dispositivo *Zynq* SoC (*System on Chip*) que incluye una sección de FPGA para desarrollo de hardware digital, una sección con un microcontrolador ARM Cortex-A9 así como canales de comunicación entre ambas secciones. En la sección FPGA se desarrolla con base a lenguajes descriptivos de hardware o equivalentes, y en la sección ARM con base a lenguajes de alto nivel como C.

Para la evaluación del desempeño del sistema se implementaron algoritmos de procesamiento de señales clasificados en tres tipos de acuerdo a su complejidad computacional. Transformaciones de pixel a pixel clasificados de baja complejidad, transformaciones de ventana (*kernel* o filtros) definidos como de complejidad media y en algoritmos de alta complejidad se trabajó con detección de bordes en ocho direcciones, detección de esquinas de Harris y transformada de Hough. Se evaluaron varios parámetros o métricas de rendimiento de la implementación de los algoritmos con resultados satisfactorios al comparar los valores de las

métricas evaluadas contra su implementación en Matlab. Se cumplió con el objetivo del cómputo en tiempo real de los algoritmos.

Se presentan también detalles técnicos de las implementaciones en el *Zynq* en sus dos secciones FPGA - ARM así como de los canales de comunicación entre ellas, detalles de la definición en Vivado (IDE de Xilinx) utilizando herramientas avanzadas como HLS Video y librerías de OpenCV. El *Zynq* está embebido en una tarjeta *ZedBoard* de Diligent, la integración a la tarjeta de la cámara de video y de la salida de VGA entre otros elementos complementan el trabajo presentado en el documento.

## **CONTENIDO**

|                                                                                               |            |
|-----------------------------------------------------------------------------------------------|------------|
| <b>LISTA DE FIGURAS.....</b>                                                                  | <b>xii</b> |
| <b>LISTA DE TABLAS.....</b>                                                                   | <b>xiv</b> |
| <b>CAPÍTULO I INTRODUCCIÓN .....</b>                                                          | <b>1</b>   |
| 1.1. Motivación .....                                                                         | 2          |
| 1.2. Delimitaciones.....                                                                      | 3          |
| 1.3. Contribuciones .....                                                                     | 3          |
| 1.4. Presentación de la tesis.....                                                            | 4          |
| 1.5. Estado del arte .....                                                                    | 4          |
| <b>CAPÍTULO II FPGA E INTEGRACIÓN EN SISTEMAS DE UN SOLO CHIP (SoC) ZYNQ Y HARDWARE .....</b> | <b>8</b>   |
| 2.1. Dispositivos FPGA.....                                                                   | 8          |
| 2.1.1 Elementos que componen a un FPGA.....                                                   | 9          |
| 2.2. Comparación del FPGA contra otras arquitecturas digitales .....                          | 10         |
| 2.3. Zynq “System on Chip” (SoC) de Xilinx.....                                               | 12         |
| 2.3.1. Unidad de sistema de procesamiento (PS) .....                                          | 14         |
| 2.3.2. Unidad de lógica programable (PL) .....                                                | 15         |
| 2.3.3. Vivado Design Suite.....                                                               | 16         |
| 2.3.4. Comunicación entre PL y PS.....                                                        | 16         |
| 2.4. Diligent ZedBoard.....                                                                   | 18         |
| 2.5. Cámara de video OV7670 .....                                                             | 20         |
| <b>CAPÍTULO III ALGORITMOS PARA PROCESAMIENTO DIGITAL DE IMÁGENES .....</b>                   | <b>26</b>  |
| 3.1. Tipos de procesamiento de imágenes digitales .....                                       | 26         |
| 3.2. Transformación RGB a escala de Grises.....                                               | 27         |
| 3.3. Detección de bordes Sobel .....                                                          | 28         |
| 3.4. Detección de esquinas por Harris.....                                                    | 28         |
| 3.5. Transformada de Hough.....                                                               | 31         |
| <b>CAPÍTULO IV DESARROLLO DEL SISTEMA EN SoC-FPGA.....</b>                                    | <b>33</b>  |
| 4.1. Diseño de módulo de configuración del sensor en VHDL.....                                | 33         |
| 4.1.1. Etapas de transmisión de datos para la configuración del sensor OV7670 .....           | 34         |
| 4.1.2. Diseño VHDL para la configuración del sensor OV7670.....                               | 36         |
| 4.2. Módulo de captura de imagen .....                                                        | 39         |
| 4.3. Diseño del módulo VGA para visualización de la imagen.....                               | 41         |

|                         |                                                                                 |            |
|-------------------------|---------------------------------------------------------------------------------|------------|
| 4.3.1.                  | Funcionamiento de la interfaz VGA .....                                         | 41         |
| 4.3.2.                  | Modulo VGA en VHDL.....                                                         | 42         |
| 4.4.                    | Integración de los módulos .....                                                | 43         |
| 4.5.                    | Implementación de algoritmos punto a punto .....                                | 46         |
| <b>CAPÍTULO V</b>       | <b>VIVADO HIGH LEVEL SYNTHESIS (HLS) .....</b>                                  | <b>49</b>  |
| 5.1.                    | Herramientas HLS.....                                                           | 49         |
| 5.2.                    | Diseño con “Vivado HLS” .....                                                   | 50         |
| 5.2.1.                  | Optimización de código y sintetización .....                                    | 52         |
| 5.3.                    | Librería OpenCV para Vivado HLS.....                                            | 56         |
| 5.3.1.                  | Librería HLS de video ( <i>HLS Video Library</i> ).....                         | 57         |
| <b>CAPÍTULO VI</b>      | <b>DISEÑO DE MÓDULOS EN VIVADO HLS PARA PROCESAMIENTO DE IMÁGENES ....</b>      | <b>59</b>  |
| 6.1.                    | Algoritmo de Sobel para Detección de Bordes Modificado a Ocho Direcciones ..... | 60         |
| 6.2.                    | Detección de esquinas por el algoritmo de Harris.....                           | 66         |
| 6.3.                    | Transformada de Hough para Detección de Líneas .....                            | 70         |
| 6.4.                    | VDMA y modificación de módulo OV7670 y VGA .....                                | 75         |
| 6.4.1                   | VDMA.....                                                                       | 75         |
| 6.4.2.                  | Adaptación del módulo ov7670_capture.....                                       | 78         |
| 6.4.3.                  | Adaptación del módulo de visualización VGA .....                                | 80         |
| <b>CAPÍTULO VII</b>     | <b>IMPLEMENTACIÓN DEL SISTEMA Y RESULTADOS .....</b>                            | <b>83</b>  |
| 7.1.                    | Integración del sistema en el SoC FPGA para procesamiento de imágenes .....     | 83         |
| 7.2.                    | Métricas Utilizadas para Evaluación del Rendimiento .....                       | 86         |
| 7.3.                    | Resultados .....                                                                | 87         |
| 7.3.1.                  | Detección de bordes en 8 direcciones .....                                      | 87         |
| 7.3.2.                  | Detección de esquinas por Harris.....                                           | 89         |
| 7.3.3.                  | Transformada de Hough para detección de líneas.....                             | 92         |
| <b>CAPÍTULO VIII</b>    | <b>CONCLUSIONES.....</b>                                                        | <b>96</b>  |
| 8.1.                    | Propuesta de trabajo futuro .....                                               | 99         |
| <b>REFERENCIAS.....</b> |                                                                                 | <b>100</b> |

## LISTA DE FIGURAS

|                                                                                                                                                                                                                                                                                     |    |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| Figura 2.1. Arquitectura general de los FPGA tradicionales [27].                                                                                                                                                                                                                    | 9  |
| Figura 2.2 Comparación de un sistema con componentes aislados (izquierda) y sistema con componentes integrados (derecha) [28].                                                                                                                                                      | 13 |
| Figura 2.3 Modelo simplificado de la arquitectura de <i>Zynq</i> [28].                                                                                                                                                                                                              | 13 |
| Figura 2.4 Sistema de procesamiento del <i>Zynq</i> [28].                                                                                                                                                                                                                           | 14 |
| Figura 2.5 Elementos que se encuentran en el FPGA del <i>Zynq</i> [28].                                                                                                                                                                                                             | 15 |
| Figura 2.6 Comportamiento de señales de la Tabla 2.1 en la transmisión de datos [16].                                                                                                                                                                                               | 17 |
| Figura 2.7 Editor de direcciones previo a síntesis en Vivado.                                                                                                                                                                                                                       | 18 |
| Figura 2.8 Diagrama de bloques de la tarjeta ZedBoard [42].                                                                                                                                                                                                                         | 20 |
| Figura 2.9 Esquema de la cámara OV7670 [43].                                                                                                                                                                                                                                        | 21 |
| Figura 2.10 Señal de sincronización horizontal [43].                                                                                                                                                                                                                                | 23 |
| Figura 2.11 Señal de sincronización vertical [43].                                                                                                                                                                                                                                  | 24 |
| Figura 2.12 Mapa de tiempos PCLK y transmisión de datos RGB565 [43].                                                                                                                                                                                                                | 25 |
| Figura 3.1 Representación del método Harris en el espacio $\lambda_1$ - $\lambda_2$ .                                                                                                                                                                                               | 30 |
| Figura 3.2(a) Espacio x-y, y (b) espacio de Hough $\rho$ - $\theta$ [49].                                                                                                                                                                                                           | 31 |
| Figura 4.1 Esquema del protocolo SCCB                                                                                                                                                                                                                                               | 34 |
| Figura 4.2 Diagrama de tiempos para la transmisión de datos mediante protocolo SCCB [50].                                                                                                                                                                                           | 34 |
| Figura 4.3 Esquema de transmisión de un byte [27].                                                                                                                                                                                                                                  | 35 |
| Figura 4.4 IP módulo encargado de configurar el sensor.                                                                                                                                                                                                                             | 38 |
| Figura 4.5 Parámetros configurables de memoria.                                                                                                                                                                                                                                     | 40 |
| Figura 4.6 Configuración de parámetros para puerto A de la memoria.                                                                                                                                                                                                                 | 40 |
| Figura 4.7 Módulo encargado de capturar datos del sensor.                                                                                                                                                                                                                           | 41 |
| Figura 4.8 Comportamiento de señales de sincronización HSYNC y VSYNC [53].                                                                                                                                                                                                          | 42 |
| Figura 4.9 Sistema integrado en FPGA.                                                                                                                                                                                                                                               | 44 |
| Figura 4.10 Esquema de sistema en FPGA.                                                                                                                                                                                                                                             | 47 |
| Figura 5.1 Diseño en Vivado HLS para la aceleración de funciones.                                                                                                                                                                                                                   | 52 |
| Figura 5.2 Ejemplos de diferentes síntesis del ciclo definido en cuadro gris, “Rolled Loop” es realizado en 4 ciclos de reloj, y se usan pocos recursos, mientras que los ejemplos desenrollados usan más recursos para la ejecución de 2 y 1 ciclo de reloj, respectivamente [59]. | 54 |
| Figura 5.3 Pipelining en un ciclo [63].                                                                                                                                                                                                                                             | 55 |
| Figura 5.4 Los ciclos separados se les aplica pipelining para que cada ciclo consecutivo pueda ejecutar sus iteraciones independientemente [59].                                                                                                                                    | 56 |
| Figura 6.1 Esquema de diseño para módulos IP en Vivado HLS                                                                                                                                                                                                                          | 59 |
| Figura 6.2 Esquema de bloques de la secuencia de datos en el algoritmo de Sobel en 8 direcciones. ..                                                                                                                                                                                | 60 |
| Figura 6.3 Tiempos de ejecución de la función principal <code>image_filter</code> .                                                                                                                                                                                                 | 63 |
| Figura 6.4 Estimación en rendimiento y uso de recursos en el módulo <code>image_filter</code> .                                                                                                                                                                                     | 64 |
| Figura 6.5 Secuencia de procesos y control de pasos generada por Vivado HLS para el módulo.                                                                                                                                                                                         | 65 |
| Figura 6.6 Estimación en rendimiento y recursos en módulos <code>image_sum</code> .                                                                                                                                                                                                 | 66 |
| Figura 6.7 Flujo de datos en algoritmo para detección de esquinas.                                                                                                                                                                                                                  | 66 |
| Figura 6.8 Función de Harris en HLS [67].                                                                                                                                                                                                                                           | 67 |
| Figura 6.9 Rendimiento estimado en módulo de Harris.                                                                                                                                                                                                                                | 69 |

|                                                                                                                         |    |
|-------------------------------------------------------------------------------------------------------------------------|----|
| Figura 6.10 Pipeline del algoritmo de Harris para detección de Harris en HLS. ....                                      | 69 |
| Figura 6.11 Diagrama propuesto para TH en HLS. ....                                                                     | 70 |
| Figura 6.12 Rendimiento estimado para algoritmo de HT. ....                                                             | 73 |
| Figura 6.13 Flujo de datos marcados por un Pipelining tipo Dataflow en HLS para la TH. ....                             | 73 |
| Figura 6.14 Jerarquía de funciones en hls::HoughLines2. ....                                                            | 74 |
| Figura 6.15 Recursos estimados por HLS en la implementación de la TH. ....                                              | 74 |
| Figura 6.16 Diagrama del IP VDMA [69]. ....                                                                             | 76 |
| Figura 6.17 Interacción del módulo IP con el PS y PL mediante las Interfaces AXI. ....                                  | 77 |
| Figura 6.18 Configuración del VDMA. ....                                                                                | 78 |
| Figura 6.19 Bloque IP para captura con comunicación AXI-Stream. ....                                                    | 79 |
| Figura 6.20 Mapa de conexiones entre los IP [71]. ....                                                                  | 81 |
| Figura 6.21. Configuración para Video Timing Controller. ....                                                           | 82 |
| Figura 6.22. Configuración del IP AXI4-Stream to Video Out. ....                                                        | 82 |
| Figura 7.1 Diagrama del sistema diseñado para procesamiento de imágenes en tipo real. ....                              | 83 |
| Figura 7.2 Imagen XTPG_BKGND_CROSS_HATCH capturada para análisis en Matlab. ....                                        | 85 |
| Figura 7.3 Modificación a sistema para evaluación de resultados. ....                                                   | 86 |
| Figura 7.4 Recursos utilizados en la implementación de detección de bordes en 8 direcciones por método de Sobel. ....   | 88 |
| Figura 7.5 Datos captados por la cámara OV7670 y desplegados por VGA a 60 cuadros por segundo. ....                     | 88 |
| Figura 7.6 Resultados en métricas de evaluación. ....                                                                   | 89 |
| Figura 7.7 Recursos utilizados en el Zynq para el detector de esquinas Harris. ....                                     | 90 |
| Figura 7.8 Detección de esquinas por Harris en tablero de ajedrez. ....                                                 | 90 |
| Figura 7.9 Comparación de resultados entre Matlab y el sistema implementado en el Zynq para detección de esquinas. .... | 91 |
| Figura 7.10 Recursos empleados en el sistema para cálculo de la TH en detección de líneas. ....                         | 92 |
| Figura 7.11 Resultados de la TH para detección de líneas. ....                                                          | 94 |

## **LISTA DE TABLAS**

|                                                                                                                    |    |
|--------------------------------------------------------------------------------------------------------------------|----|
| Tabla 2.1 Señales utilizadas en la interfaz AX4I-Stream [39]. .....                                                | 17 |
| Tabla 2.2 Pines del sensor OV7670. ....                                                                            | 22 |
| Tabla 7.1 Velocidades de procesamiento para el algoritmo de Sobel en ocho direcciones. ....                        | 89 |
| Tabla 7.2 Diferencia de distancia entre esquinas encontradas en Matlab y el algoritmo Implementado en el Zynq..... | 91 |
| Tabla 7.3 Velocidades de procesamiento para el algoritmo Harris.....                                               | 92 |
| Tabla 7.4 Velocidades de procesamiento para el algoritmo de la TH para detección de líneas.....                    | 94 |

## CAPÍTULO I

### INTRODUCCIÓN

En la actualidad el procesamiento de imágenes es uno de los principales métodos empleados para la solución de problemas en diferentes áreas, gracias a su facilidad de interpretación visual, ayudando a comprender el problema y dar solución a este mismo mediante métodos matemáticos como convolución y correlación. En general los problemas abordados en el área de procesamiento de imágenes son usados por un sistema de visión que captura imágenes que son procesados por dichos algoritmos, tales como: detección de color, operaciones morfológicas ya sea erosión, dilatación, detección de bordes, detección de esquinas, entre otras. Los algoritmos ayudan a eliminar información no deseada, así como también resaltar información que sea de uso en el sistema de visión. Existen algoritmos que transforman la información, siendo capaces de obtener características que el ser humano no es capaz de detectar debido a delimitaciones físicas que perturban la admisión de información, entre ellas: enfermedades genéticas, ancho de banda, así como fenómenos naturales como ilusiones ópticas que impiden diferenciar entre distintos objetos.

Por lo general, los sistemas de visión son implementados en plataformas con ejecuciones de instrucciones secuenciales tal como una Unidad de Procesamiento Central (CPU), ofreciendo la facilidad de uso en esta área gracias a librerías existentes que simplifican el diseño como lo es OpenCV (*Open Computer Vision*). No obstante aplicaciones en las que se tiene una gran cantidad de datos que integran una imagen o algoritmos con operaciones iterativas, hacen uso sistemas que permiten la ejecución en paralelo como un FPGA (*Field Programmable Gate Array*) o una GPU (*Graphics Processing Unit*). Las plataformas con ejecución en paralelo aminoran la carga de datos y operaciones que comprometen los recursos del CPU (*Central Processing Unit*).

Para caso de estudio, en esta tesis se estableció el uso del sistema en un solo chip, SoC (*System On Chip*), llamado Zynq de Xilinx incluido en la tarjeta de desarrollo ZedBoard, en el cual se implementa un sistema de adquisición de imágenes con base a la cámara OV7670 y

algoritmos de procesamiento diseñados con Vivado HLS, quien permite hacer uso de la librería OpenCV de manera optimizada para una síntesis en lenguajes descriptivos en VHDL (*Very High Speed Integrated Circuit Hardware Description Language*) o Verilog. Facilitando la evaluación del sistema diseñado, así como los algoritmos empleados con Matlab. El sistema hace uso del CPU y FPGA para un diseño en conjunto, la mayoría del algoritmo está basado en el FPGA debido a que se plantea un sistema con desplegado de resultados en tiempo real mediante el periférico VGA (*Video Graphics Array*).

### **1.1. Motivación**

El procesamiento de imágenes se utiliza en diversas áreas tales como: comunicaciones, automóviles, defensa, seguridad, medicina, entre otras. Todas estas áreas manejan tiempos de ejecución de instrucciones o procesos como parte esencial ya que se busca el tiempo real, por lo tanto, el uso de arquitecturas basadas en FPGA y arquitecturas basadas en procesadores ARM Cortex trabajando en conjunto atacan dicho problema aprovechando la ejecución en paralelo que ofrece el FPGA y el control de software en el procesador ARM Cortex del sistema SoC Zynq 7000 de Xilinx.

El desarrollar componentes lógicos definidos en el FPGA y usar su ejecución en paralelo mediante instancias que permitirán en principio generar ahorros importantes en tiempo de ejecución. Los componentes desarrollados pueden ser de acceso público y adaptables a diferentes aplicaciones, contar con ellos ayudará a las áreas antes mencionadas reduciendo tiempos de diseño ya que solo se instanciarán componentes creados para dicho procesamiento en el FPGA.

Debido a la baja incursión en los sistemas SoC por su complejidad dada la arquitectura basada en FPGA, surge la necesidad de desarrollar información que permita la comprensión de esta tecnología para reducir los tiempos de aprendizaje de los futuros usuarios de esta tecnología.

Y por último este proyecto también viene motivado por la explotación y la investigación de las nuevas herramientas para el desarrollo de hardware desde interfaces y lenguajes de alto nivel, dejando atrás la programación de hardware sobre lenguajes descriptivos.

## 1.2. Delimitaciones

Esta tesis tendrá un énfasis acerca del funcionamiento del sistema *Zynq* en la tarjeta *ZedBoard* en relación con las limitaciones que impone un SoC. Por lo tanto, no tendrá como objetivo encontrar la mejor optimización de un algoritmo para el procesamiento digital de imágenes, solo uno que funcione a un nivel satisfactorio y que pueda ser comparado con Matlab. Se proporcionará una visualización adaptada máximo a 60 cuadros por segundo debido al tipo de monitor a utilizar, sin embargo, se considerará la máxima tasa de procesamiento del sistema sintetizable sin una demostración visual. Una delimitación adicional son los recursos con los que cuenta el SoC a utilizar la cual se podrá observar en el algoritmo de la Transformada de Hough para detección de líneas. Además, la falta de experiencia previa con herramientas de diseño como lo es Vivado HLS.

## 1.3. Contribuciones

Dentro de las aportaciones que se obtienen con este trabajo de tesis se encuentran:

- Creación de módulos para la obtención y configuración de la cámara OV7670, como también el módulo para visualizar los resultados en tiempo real por medio del periférico VGA con el que cuenta la tarjeta *ZedBoard*.
- Sinterización de módulos para: detección de bordes en ocho direcciones basado en Sobel y detección de esquinas por Harris y por medio de la Transformada de Hough.
- Plantilla descriptiva en lenguaje descriptivo del protocolo de comunicación AXI4-Stream para transmisión de datos a altas velocidades.
- Integración de las arquitecturas FPGA-CPU para la implementación y control de datos respectivamente en el sistema *Zynq*.
- Evaluación y comparación de resultados en Matlab de los algoritmos diseñados e implementados en el sistema *Zynq*.

Se proponen metodologías utilizando la ejecución en paralelo para obtener un mejor desempeño, logrando una tasa de procesamiento de 80 cuadros por segundo con una frecuencia base de 25 MHz y un procesamiento hasta de 623 cuadros procesados por segundo aproximados a una frecuencia base de 200 MHz en el caso del algoritmo de Sobel. Demostrando la velocidad de procesamiento con la que cuenta el *Zynq* ante

Además, se exponen la herramienta de Vivado HLS que es usada para la elaboración de diseños con lenguajes de alto nivel, haciendo entendible y fácil el diseño en FPGA.

#### **1.4. Presentación de la tesis**

En este trabajo de tesis se describe el desarrollo de la plataforma FPGA SoC para el análisis de algoritmos de procesamiento digital de imágenes. El segundo capítulo trata acerca de una exposición de conceptos, características y elementos que componen el hardware usado como: tarjeta *ZedBoard* y la cámara OV7670. En el capítulo 3 se describen los algoritmos seleccionados para su previa implementación. En el capítulo 4 se propone una metodología para el sistema de adquisición de imágenes incluyendo los módulos creados paso a paso.

Debido a el uso de recursos el capítulo 5 da una introducción a Vivado HLS, así como sus componentes y enfoques de diseño que son utilizados en este ambiente de desarrollo. Después de la descripción de Vivado HLS, en el capítulo 6 se hace uso de esta herramienta para la implementación de los algoritmos. También se incluye el diseño de guardado en VDMA de los datos capturados por el módulo de captura OV7670 con adaptación al protocolo AXI4-Stream.

Se muestra en el capítulo 7 la implementación final de cada uno de los algoritmos con el sistema de captura, dando paso a la toma de resultados que son evaluados y comparados con Matlab en el capítulo 8. El noveno y último capítulo expone las conclusiones obtenidas durante la realización de este proyecto.

#### **1.5. Estado del arte**

Los trabajos donde se implementa el algoritmo de Sobel, que es uno de los más usados en el procesamiento de imágenes, se encuentran aquellos que no hacen uso de HLS para su síntesis,

por ejemplo, en [1]–[4], quienes realizan todo el módulo utilizando lenguajes de descripción de hardware (VHDL o Verilog), optimizado para el rendimiento y un mejor uso de recursos. Pero existen trabajos que realizan el mismo algoritmo pero con uso de HLS como los mostrados en [5]–[7] que en comparación con recursos ¿computacionales? tienden a ser más demandantes que los generados e implementados desde lenguaje descriptivo. La mayoría de los trabajos hace uso de las herramientas HLS debido al diseño rápido. No obstante, estas herramientas son usadas para aplicaciones complejas que permitan trabajar con los *Zynq* ahorrando tiempo de diseño y aprovechando la partición de tareas en el FPGA y CPU. En trabajos encontrados en la literatura está la aplicación de los sistemas *Zynq* y uso de herramientas HLS para el diseño de aplicaciones para el reconocimiento de señales de tráfico, donde se realizan filtrados y transformaciones a las imágenes para posteriormente pasar a un sistema de clasificación. Las operaciones son demandantes por lo que particionar tareas en un FPGA es de gran ayuda. En [8] se muestra una implementación en FPGA *Zynq-7000* de la transformada de Hough que es evaluada en aplicaciones para la detección de señales de tráfico y análisis de imágenes aéreas. El sistema diseñado opera con una frecuencia de 50 MHz y los resultados son mostrados por medio de una salida VGA, donde dichos resultados superaron en velocidad hasta 4.5 veces mayor a la velocidad de un CPU i7 de gama alta. Una conclusión de este trabajo es la complejidad en recursos que maneja la Transformada de Hough debido a que los autores hacen uso de la mayoría de los recursos del *Zynq* en el módulo de la Transformada (*slices* 71%, BRAM 100%, DSP 73%); también en [9] los autores usan la TH. En el artículo [10] se realiza una segmentación de señales de camino por medio de una clasificación de distancia mínima, donde la imagen pasa por filtros bidimensionales como lo es el gaussiano y un filtro de mediana. Los filtros fueron diseñados en el ambiente VHDL. Los autores muestran rendimientos del 98% en segmentación. Otro trabajo donde se hace detección de señales de tráfico en un *Zynq* se muestra en [11], donde los autores desarrollan la mayor parte del sistema en el ARM del *Zynq*, al cual, se le integra un sistema operativo que les permite el uso de librerías de OpenCV. El hacer uso de un sistema operativo hizo que los autores manejaran una tasa de procesamiento de 5 imágenes por segundo. En [12] los autores optan por una segmentación de color y plantillas predefinidas para el ahorro de procesamiento. El uso de la partición de tareas para la detección de señales se ve mostrado por los autores en los trabajos anteriores, al igual que en [13] donde se separan las tareas

demandantes a la parte PL de *Zynq*, donde en el sistema toma la imagen desde el PS por medio del protocolo UART, posteriormente se envían los datos por comunicación AXI al PL para su procesamiento mediante módulos IP de filtros morfológicos y devuelve los datos de los candidatos a señales de tráfico al PS nuevamente. El sistema es lento debido a la comunicación UART, pero si se manejara el sistema de captura desde el PL mediante otro puesto de comunicación o, vía un sensor sería más rápido, debido a que en este diseño se tarda 96.5 ms en procesar una imagen y el IP diseñado con HLS tarda 28.7 ms.

Una aplicación interesante del sistema *Zynq* es la mostrada en [14] donde los autores realizan un sistema que monitorea la atención de los usuarios al conducir, la imagen es adquirida por medio de un *Kinect* y procesada en el ARM al cual se le instaló un sistema operativo donde se realiza el procesamiento y provocando que la tasa de cuadros procesados por segundo decaiga. El usar un procesador no quiere decir que se disminuya siempre la velocidad de procesamiento, en [15] se utiliza una repartición de tareas procesadores simulados en FPGA para el algoritmo *K-means*. En [16] y [17] también se implementa el algoritmo *K-means* con ayuda de un HLS, lo que demuestra la gran facilidad de sintetizado de algoritmos demandantes en interacciones computacionales.

Se muestra en el artículo [18] el diseño de un sistema de análisis de imágenes de resonancia magnética, donde las imágenes son pasadas por filtros y caracterización de tumores. En este trabajo se usa *Xilinx System Generator* (XSG) que es una herramienta de síntesis de alto nivel que permite trabajar con Matlab y Simulink, el autor hace uso de esta herramienta para crear los módulos de filtrado y caracterización obteniendo un bajo costo en recursos utilizados del FPGA (XC3S500E-FG320) a una frecuencia de operación de 50 MHz, no obstante, el diseño no presenta una implementación en tiempo real.

En aplicaciones de algoritmos para el seguimiento de objetos en secuencias de imágenes se encuentra el artículo mostrado en [19], donde se utiliza una tarjeta *ZedBoard* obteniendo un tiempo de 768 milisegundos por cuadro procesado, el diseño utiliza el procesador ARM y el FPGA incluidos.

En [20] se muestra el uso del *Zynq* de la tarjeta *ZedBoard* usada en otra área de imágenes, como lo es la compresión de imágenes, en donde se realiza una compresión JPEG 2000 utilizando solo el 5% de los recursos del sistema *Zynq*, utilizando un diseño basado en HLS usando Matlab.

Los trabajos [21]–[24] muestran el uso de sistemas *Zynq*, FPGA en los cuales se implementan sistemas basados en metodologías de inteligencia artificial, demandando el análisis de un gran cantidad de datos. Demostrando los sistemas FPGA *Zynq* como una herramienta rápida en ejecuciones y un bajo consumo de energía ante otros SoC [25].

## CAPÍTULO II

### FPGA E INTEGRACIÓN EN SISTEMAS DE UN SOLO CHIP (SoC) ZYNQ Y HARDWARE

A continuación, se describen los dispositivos reconfigurables utilizados en la implementación del sistema de procesamiento de imágenes. Se inicia con la descripción del dispositivo FPGA (*Field Programmable Gate Array*, por sus siglas en inglés), para terminar con los detalles de la tarjeta *ZedBoard* utilizada para el desarrollo con el dispositivo SoC-FPGA embebido y los periféricos que complementan el sistema como lo es la cámara de video para la entrada de datos.

#### 2.1. Dispositivos FPGA

Los FPGA fueron diseñados en la década de 1980 por Ross H. Freeman, los cuales están basados en los llamados Dispositivos Lógicos Reprogramables (*PLD*, por sus siglas en inglés) [26]. Los PLD se configuran (programan) para diseñar circuitos digitales, desde simples circuitos con compuertas lógicas hasta sistemas digitales complejos. Los PLD han evolucionado desde su primera aparición, los primeros dispositivos contenían el equivalente a miles de compuertas lógicas para llevarlos a alguna aplicación y en la actualidad se encuentran dispositivos con millones de compuertas disponibles. Pasa lo mismo con las entradas y salidas, en los dispositivos antiguos se pueden manejar pocas salidas caso contrario en los dispositivos modernos, con cientos de entradas y salidas. Los primeros PLD se configuraban “quemando” una matriz de fusibles y eran programables una sola vez (OTP, por sus siglas en inglés).

Los PLD se clasifican en tres tipos:

- Dispositivos lógicos programables simples (SPLD, por sus siglas en inglés).
- Dispositivos lógicos programables complejos (CPLD, por sus iniciales en inglés).
- Arreglos de compuertas programables en campo (FPGA)

Cabe destacar que las diferencias entre los CPLD y FPGA no son tan notorias, por lo que, a los dos se les conoce como dispositivos lógicos de alta capacidad (HCPLD, por sus siglas en inglés).

Los FPGA consisten de diversos módulos lógicos, programables e independientes que pueden interconectarse para formar funciones lógicas más grandes (en función de la complejidad del diseño). Por lo general, los FPGA trabajan con el método de tabla de búsqueda (LUT, por sus siglas en inglés) para crear diversas funciones lógicas. Las entradas y salidas se configuran en módulos que pueden funcionar de manera bidireccional. En la Figura 2.1 se muestra el diagrama general de un FPGA donde se presenta a grandes rasgos el posicionamiento de cada una de las características o componentes distintivas.

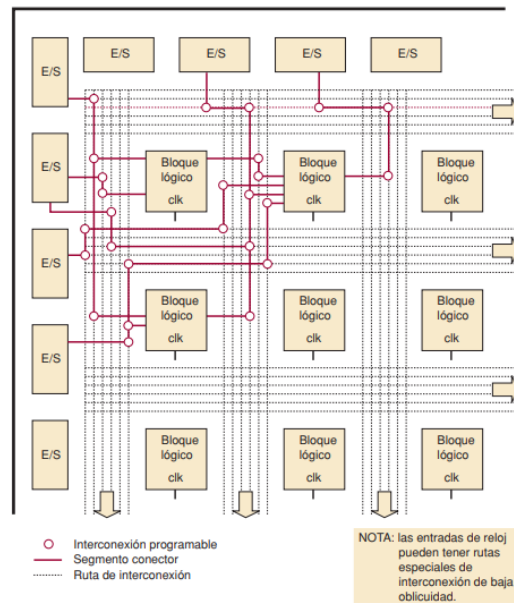


Figura 2.1. Arquitectura general de los FPGA tradicionales [27].

### 2.1.1. Elementos que componen a un FPGA

Los FPGA se estructuran principalmente en Bloques Lógicos Reconfigurables (CLB, por sus siglas en inglés). Las arquitecturas de un CLB no cuentan con un estándar estricto entre los diferentes fabricantes de FPGA [28], [29]. A continuación, se describen los componentes de la arquitectura de un FPGA de la serie 7 de Xilinx, serie usada en este proyecto.

**Flip-Flops (FF):** Un *Flip-Flop* es un dispositivo de almacenamiento, que solo puede tener dos estados, equivalente a un bit de información. En la serie 7 Xilinx, cuatro de los ocho FF incluidos en una *slice* (rebanada) pueden configurarse como *Flip-Flops* estándar

(almacenamiento síncrono), mientras que los cuatro restantes pueden configurarse como *Flip-Flops* estándar o como *latches* (almacenamiento asíncrono) [30]. Sin embargo, el uso de los *latches* generalmente no se considera una buena práctica, ya que conducen a problemas de sincronización.

**Rebanada o Slice:** Es una unidad interna del CLB. Cada CLB consta de dos *slices* que pueden ser de dos tipos SLICEM y SLICEL. Cada *slice* contiene cuatro LUT, ocho *Flip-Flops* (FF), una red de interconexiones lógicas y tres tipos de multiplexores (*mux*). En un chip, existen el doble de *slices* tipo SLICEL que de tipo SLICEM. Esto se debe al hecho de que las *slice* tipo SLICEM proporcionan una funcionalidad adicional a la normal, también puede configurarse para que sus LUT internas puedan funcionar como registros de desplazamiento o como almacenamiento de datos, lo que permite que el FPGA funcione como memoria distribuida [30]. Una sugerencia es asociar la memoria con la SLICEM (segmento M) y a la lógica con la SLICEL (segmento L).

**Tabla de búsqueda (LUT):** En las LUT se implementa la lógica del FPGA y consta de cierto número de entradas y una salida cuyo valor se configura para todos los conjuntos de entradas posibles. La LUT consiste en un bloque de RAM, cuyos índices están asociados con las entradas de la LUT [31].

La tecnología para la configuración (“programación”) en los FPGA va desde SRAM, Flash y antifusible, de las tres mencionadas la más común es la tecnología SRAM, de tipo volátil, por tanto, pierde su configuración al no estar encendido el dispositivo. Al encender el FPGA es necesario volver a configurarlo [27].

## 2.2. Comparación del FPGA contra otras arquitecturas digitales

Una forma de disminuir el tiempo de ejecución del código es hacer uso de ejecuciones en paralelo lo que permite que se realicen varios cálculos simultáneamente, suponiendo que el problema o código se puede dividir en varios subprocesos. La ejecución paralela viene en diferentes formas: nivel de bits, nivel de instrucción, y nivel de datos y tareas. Dependiendo del hardware en uso, no siempre todas estas acciones son posibles de implementar. Una ventaja de los FPGA es la posibilidad de implementar ejecuciones en paralelo [32].

Las unidades de procesamiento central (CPU, por sus siglas en inglés) se han encontrado con problemáticas que van desde el consumo de energía, aumento en las frecuencias de operación las cuales son mitigadas con la adición de núcleos en los chips. Por lo tanto, para usar los núcleos en el CPU de la manera más eficiente, es necesario desarrollar programas con partes que se ejecuten en paralelo, aumentando así la velocidad de ejecución [32].

Una Unidad de Procesamiento Gráfica (GPU, por sus siglas en inglés) es un procesador diseñado específicamente para realizar operaciones gráficas. Estos cálculos pueden ser de dos y tres dimensiones, pero el objetivo principal de la GPU es generar gráficos en 3D [24]. Los FPGA y las GPU son tecnologías competidoras en lo que respecta a la informática *embebida* de alto rendimiento aplicada, por ejemplo, aplicaciones de procesamiento digital de señales e imágenes[33]-[35].

Los FPGA tienen algunas ventajas de rendimiento frente a las GPU, ya que son implementaciones de algoritmos en hardware y son en general más rápidas que aquellas en software [33]. Los FPGA también son más deterministas, es decir, es decir, relacionan una misma salida partiendo de un estado definido inicialmente (no se involucra la aleatoriedad), una característica que es crucial en los sistemas en tiempo real, en cambio los usuarios de GPU compensan esto considerando el peor tiempo de ejecución en sus aplicaciones [33]. Al representar gráficos en 3D, los cálculos con números decimales son comunes, por lo tanto, las GPU están diseñadas para funcionar bien en operaciones de punto flotante [24], al acomodar procesadores de punto flotante de hardware nativo. Este aspecto hace que las GPU sean adecuadas para aplicaciones de procesamiento de señales con punto flotante [33].

Como se ha mencionado, las GPU son especialistas en hacer cálculos, se puede agregar que las GPU son rápidas gracias al ancho de banda de memoria que manejan [34] aplicando técnicas como el acceso directo a memoria (DMA, por sus siglas en inglés), esto es, transmisión de un alto volumen de datos a la GPU sin consumir ciclos de reloj [33].

Las GPU son fáciles de programar utilizando lenguajes de alto nivel e interfaces de programación de aplicaciones, resumiendo los detalles del hardware [34]. Dado que un FPGA tiene una estructura de hardware parecida a un ASIC (Circuito Integrado de Aplicación

Específica) pero reconfigurable, los diseñadores al implementar un algoritmo específico pueden decidir el hardware de módulo a módulo, intercambiar recursos y rendimiento eligiendo el nivel adecuado de paralelismo. Usando un FPGA, se pueden eliminar las ineficiencias que vienen con el modelo de ejecución de Von Neumann y, además, se puede lograr un mejor rendimiento y eficiencia energética. Los FPGA tienen una ventaja frente a las GPU en la medición del rendimiento, que consiste en una mejora del 20 al 25% [36]. No obstante, el avance de la tecnología y la incrustación de unidades como módulos de operación de punto flotante en los FPGA más recientes, hacen que el consumo energético quede a la par de las nuevas GPU. Aunque en los FPGA los proveedores proporcionan bibliotecas de aplicaciones para procesamiento en forma de módulos IP (Propiedad Intelectual), desarrollados en un Lenguaje Descriptivo de Hardware de Circuito Integrado de Muy Alta Velocidad (VHDL) o Verilog [26], [27], [32]. Crear un diseño desde cero, es una tarea laboriosa y por lo tanto, costosa [28],[34].

En resumen, las GPU y los FPGA tienen sus diferentes ventajas y desventajas. Entonces, cuando se enfrenta el problema de elegir a uno de ellos para una aplicación específica, se debe evaluar la naturaleza de la tarea en cuestión y sus requisitos. En [37] se puede observar una comparación entre el uso de CPU, FPGA y GPU K40 realizando una clasificación de máxima verosimilitud. Los autores concluyen que los FPGA pueden lograr el mejor rendimiento en comparación con el clúster de CPU convencional y los GPU K40, además, de que los FPGA son más eficientes energéticamente.

### 2.3. Zynq “System on Chip” (SoC) de Xilinx

Los dispositivos SoC, “*System on Chip*”, por lo general se refieren a circuitos integrados de aplicación específica (ASIC, por sus siglas en inglés). Pueden contener componentes digitales, analógicos y bloques para la implementación de conversiones de señales digitales a analógicas y viceversa. Los sistemas SoC pueden combinar todos los aspectos de diferentes sistemas lógicos en uno, esto para la reducción de costos monetarios, energía y espacio físico sin perder el poder de procesamiento y la lógica de alta velocidad, la Figura 2.2 muestra dicha combinación de elementos. Los ejemplos más básicos de este tipo de SoC se encuentran en PC, tabletas, teléfonos, entre otros [28].

Los dispositivos SoC se han popularizado principalmente por parte de los desarrolladores de núcleos de procesamiento ARM, debido a la flexibilidad que ofrecen sus microprocesadores. En este trabajo se utilizó un SoC diseñado por Xilinx, denominado *Zynq*, el cual es la combinación de dos elementos estructurales: un procesador ARM Cortex A9 y arreglos de compuertas programables de campo (FPGA). En los dispositivos *Zynq*, el ARM Cortex A9 es un procesador capaz de ejecutar sistemas operativos completos como Linux, mientras que la lógica programable se basa generalmente en una arquitectura del FPGA de la serie 7 de Xilinx. Toda la arquitectura se perfecciona con interfaces de comunicación tipo AXI (*Advanced eXtensible Interface*), a través de los periféricos entre la unidad de Sistema de Procesamiento (PS, por sus siglas en inglés) y la unidad de Lógica Programable (PL, por sus siglas en inglés) como se muestra en la Figura 2.3 [28].

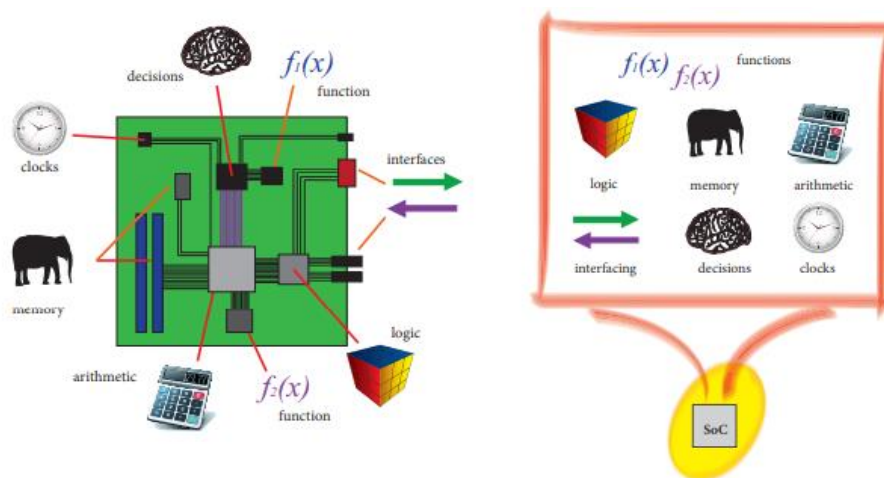


Figura 2.2 Comparación de un sistema con componentes aislados (izquierda) y sistema con componentes integrados (derecha) [28].

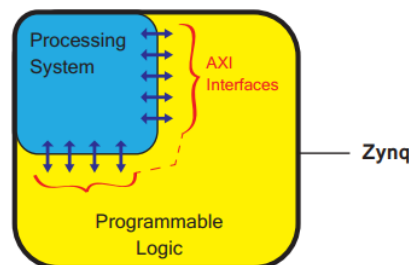


Figura 2.3 Modelo simplificado de la arquitectura de *Zynq* [28].

### 2.3.1. Unidad de sistema de procesamiento (PS)

Todos los sistemas *Zynq* de Xilinx cuentan con la misma arquitectura, un procesador ARM Cortex A9 de dos núcleos y un procesador Xilinx MicroBlaze. El procesador MicroBlaze se forma a partir de una estructura lógica programable, esto quiere decir, que el procesador MicroBlaze es simulado en un sistema digital en la parte PL del *Zynq*. Podemos tener acceso a cualquier de los dos procesadores dependiendo de la complejidad de la aplicación ya que el MicroBlaze no es un procesador dedicado [29].

El sistema de procesamiento no está solo enfocado al procesador ARM, todos los recursos asociados en esta partición que conforman una unidad de procesamiento de la aplicación (APU, por sus siglas en inglés) forman parte del PS como se observa en la Figura 2.4. La APU contiene los dos núcleos de procesamiento del ARM, una unidad de gestión de memoria (MMU), una caché de nivel 1 dividida en instrucciones y datos, una memoria caché nivel 2 y una unidad de punto flotante (FPU) en cada uno de los dos núcleos. Como se puede observar el dispositivo cuenta con recursos aparentemente poderosos para llevar a cabo tareas complejas [28].

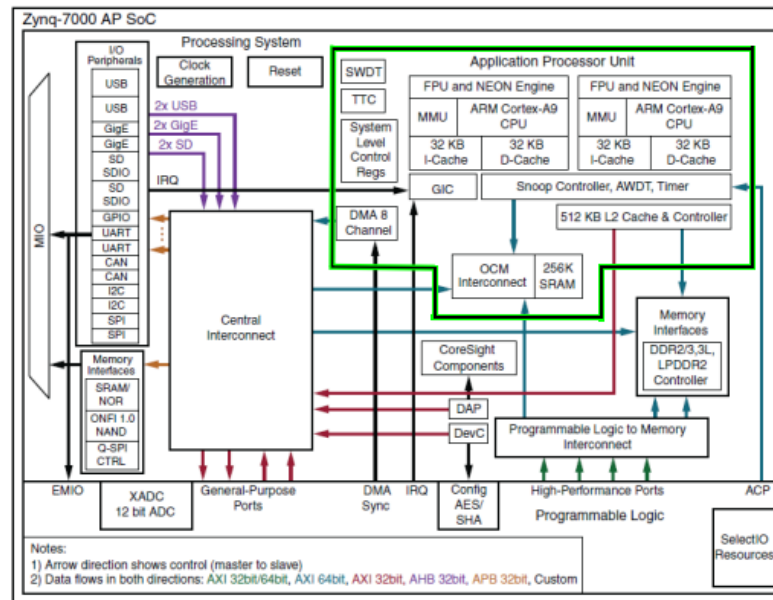


Figura 2.4 Sistema de procesamiento del *Zynq* [28].

Todos los controladores de periféricos del sistema PS están conectados como esclavos a través de la interconexión AMBA (Arquitectura Avanzada de Bus para Microcontroladores) y

tienen registros de control que se pueden leer o escribir, y que se pueden direccionar en el espacio de memoria del procesador. La sección PL también está conectada como esclava a PS a través de la interconexión AMBA.

### 2.3.2. Unidad de lógica programable (PL)

Se retoma el concepto de FPGA mencionado en la sección 2.1 para describir la unidad de lógica reprogramable PL con que cuenta el dispositivo *Zynq*. La estructura PL comprende de bloques CLB y bloques de (E/S) Entradas/Salida (IOB) para la interfaz. Los CLB se encargan de implementar funciones lógicas, funciones de memoria, o de lógica compleja. Como se mencionó en secciones anteriores, los CLB cuentan con *slices* que tienen recursos como FF y LUT, útiles para implementar lógica secuencial o combinatoria. También se encuentra una matriz de interconexiones entre los CLB o los elementos dentro del propio CLB. La unidad PL se basa en las familias de FPGA Artix-7 y Kintex-7 de Xilinx, su estructura se muestra en la Figura 2.5 [28].

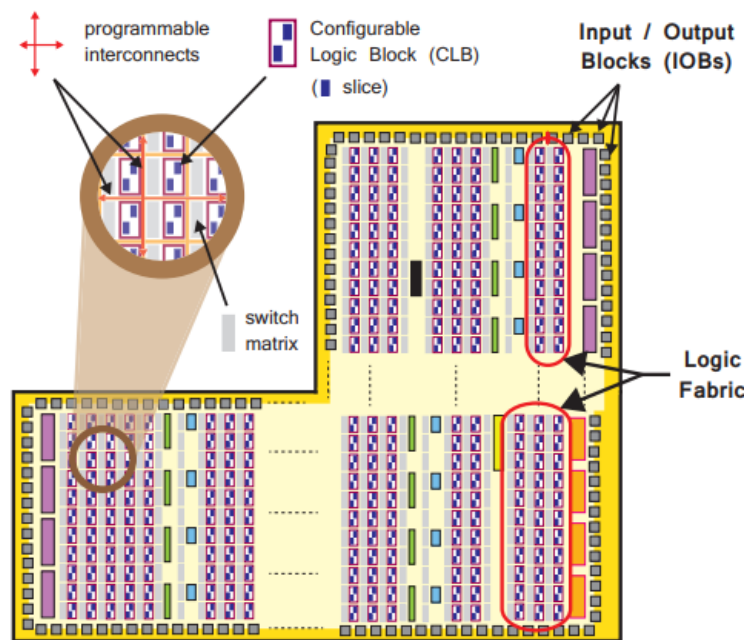


Figura 2.5 Elementos que se encuentran en el FPGA del *Zynq* [28].

### 2.3.3. Vivado Design Suite

El *Zynq* es un producto Xilinx, razón por la cual se utilizaron sus propias herramientas de diseño en este proyecto: Vivado Design Suite. El paquete de diseño Vivado contiene programas para el desarrollo de PL y PS de *Zynq*, para este proyecto se usó la versión 2018 de Vivado.

Uno de esos paquetes en Vivado Design Suite es Vivado IP Integrator (denominado simplemente Vivado) es un programa para el desarrollo de FPGA con el uso de núcleos HDL e IP, permitiendo trabajar de una manera más intuitiva debido el uso de bloques en el diseño. También contiene herramientas para depurar y programar dispositivos FPGA. Vivado gestiona las direcciones de IP mapeadas en memoria, así como los archivos de configuración para el sistema de procesamiento, que configura las frecuencias de reloj, las interrupciones y las interconexiones PS-PL. Todos estos ajustes se pueden configurar a través de menús en Vivado, evitando la tarea de configurar registros manualmente en el *Zynq*.

### 2.3.4. Comunicación entre PL y PS

Muchos bloques de los diseños realizados en la parte PL pueden comunicarse con la parte PS, así como entre sí, a través de una interfaz de bus llamada Interfaz Extensible Avanzada (AXI) mencionada anteriormente, y proviene del protocolo ARM AMBA [38]. La comunicación AXI se puede hacer con tres protocolos de capacidades diferentes. El comportamiento y la configuración general de un bloque de IP a menudo se controlan a través de un bus AXI4-Lite, adecuado para transferencias de datos mapeados en memoria de bajo rendimiento (es decir, registros de estado de lectura/escritura). El envío y/o recepción de datos más grandes, como una imagen, a menudo se realiza a través de una interfaz AXI4 estándar, más adecuada para operaciones mapeadas en memoria de alto rendimiento. La tercera interfaz *AXI4-Stream*, se utiliza para enviar continuamente una gran cantidad de datos a alta velocidad, por ejemplo, transmisión de video [39]–[41].

Se hace hincapié en la interfaz *AXI4-Stream* debido a que es la usada en este documento de tesis para la transmisión de datos. La interfaz se basa en cinco señales mostradas en la Tabla 2.1 indicando la funcionalidad de cada una. En la Figura 2.6 se observa el comportamiento de las

señales en la transmisión de datos de una secuencia de video, información adicional sobre el protocolo AXI se encuentra en el manual de usuario [41].

Tabla 2.1 Señales utilizadas en la interfaz AX4I-Stream [39].

| Señal | Controlado por | Función                                   |
|-------|----------------|-------------------------------------------|
| DATA  | Maestro        | Datos de video (cualquier número de bits) |
| VALID | Maestro        | Validación                                |
| LAST  | Maestro        | Término de línea (EOL)                    |
| USER  | Maestro        | Comienzo de cuadro (SOF)                  |
| READY | Esclavo        | Listo para recibir datos                  |

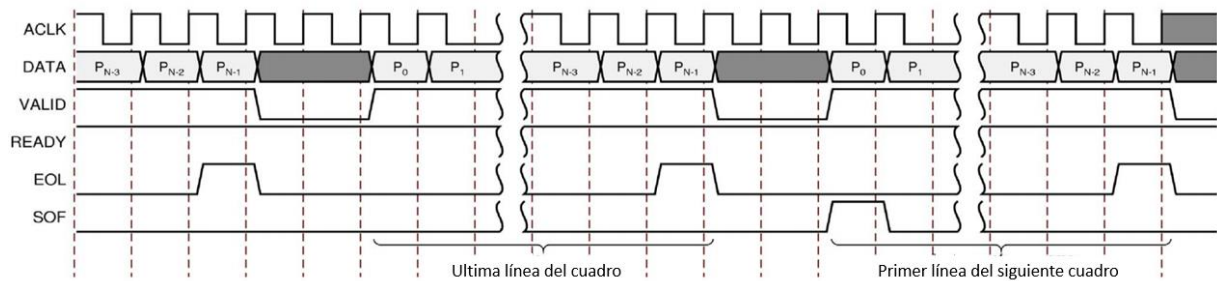
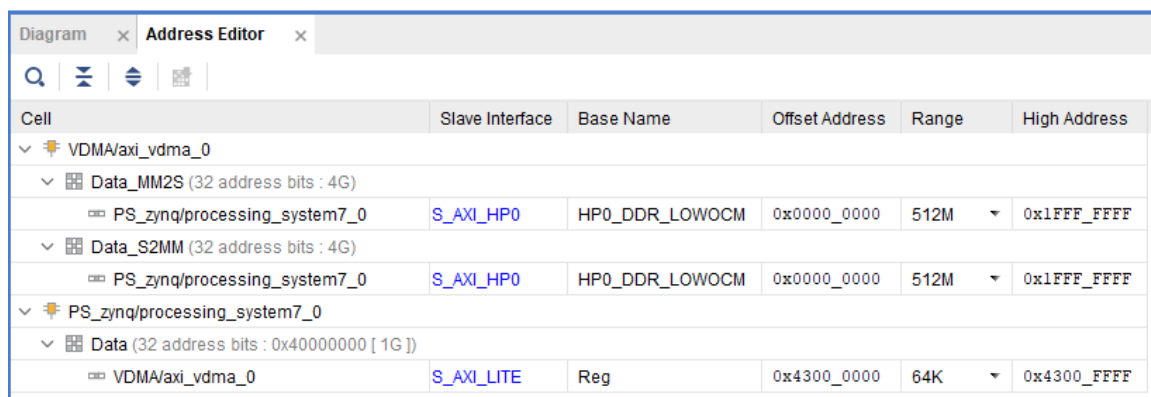


Figura 2.6 Comportamiento de señales de la Tabla 2.1 en la transmisión de datos [16].

La comunicación mediante los protocolos AXI indica que los bloques IP diseñados con una interfaz AXI4-Lite utilizan la memoria compartida entre la PS y PL para su comunicación. Esto significa que cada bloque de IP necesita un espacio de dirección que se asigna en Vivado antes de la síntesis (Figura 2.7). La transferencia de datos del bus AXI4-Lite generalmente se realiza a través de algún tipo de DMA que luego alimenta los bloques que utilizan la información. En nuestro proyecto, se utilizó principalmente el DMA de Video (VDMA), ya que está especializado para enviar secuencias de imágenes. Normalmente, los buses de control a través de AXI4-Lite obtienen un pequeño espacio de 64 kB para E/S de registro simple, mientras que la interfaz AXI4 y AXI4-Stream pueden subir a niveles de gigabytes si es necesario. Este espacio de direcciones se puede usar desde los programas que se ejecutan en la PS para configurar los ajustes de IP o transferir imágenes/datos desde o hacia el PL. Cuando finalmente se genera el flujo de bits después de la síntesis y la implementación, Vivado puede exportar un archivo de especificación de hardware que contiene todas las direcciones utilizadas en el diseño. Esta

especificación se puede importar y usar en el *Kit de Desarrollo de Software* (SDK) proporcionado por Xilinx.



| Cell                                         | Slave Interface | Base Name      | Offset Address | Range | High Address |
|----------------------------------------------|-----------------|----------------|----------------|-------|--------------|
| ▼ VDMA/axi_vdma_0                            |                 |                |                |       |              |
| ▼ Data_MM2S (32 address bits : 4G)           |                 |                |                |       |              |
| PS_zynq/processing_system7_0                 | S_AXI_HP0       | HP0_DDR_LOWOCM | 0x0000_0000    | 512M  | 0x1FFF_FFFF  |
| ▼ Data_S2MM (32 address bits : 4G)           |                 |                |                |       |              |
| PS_zynq/processing_system7_0                 | S_AXI_HP0       | HP0_DDR_LOWOCM | 0x0000_0000    | 512M  | 0x1FFF_FFFF  |
| ▼ PS_zynq/processing_system7_0               |                 |                |                |       |              |
| ▼ Data (32 address bits : 0x40000000 [ 1G ]) |                 |                |                |       |              |
| VDMA/axi_vdma_0                              | S_AXI_LITE      | Reg            | 0x4300_0000    | 64K   | 0x4300_FFFF  |

Figura 2.7 Editor de direcciones previo a síntesis en Vivado.

El SDK de Xilinx cuenta con herramientas para escribir controladores y programas C/C++ y así controlar el comportamiento de la FPGA. La especificación de hardware importada de Vivado se utiliza para formar un proyecto de plataforma de hardware y con él iniciar la placa. Se puede crear un Paquete de Soporte de Placa (BSP, por sus siglas en inglés) si se usa un sistema operativo (SO) independiente de Xilinx, que contiene controladores para dispositivos en la placa como I2C y USB, etc.

Finalmente, un usuario puede crear su proyecto de aplicación personalizado, por tanto, en este proyecto de tesis se optó por no usar un SO para realizar los procesamientos debido al bajo rendimiento obtenido de acuerdo a las conclusiones de los trabajos descritos en [11], [13], [14] y [37], quienes utilizaron el CPU (con o sin SO) a máxima frecuencia de operación para realizar procesamientos, indicando que esto no es lo más óptimo.

## 2.4. Diligent ZedBoard

El nombre de la Tarjeta de desarrollo *ZedBoard* se deriva de ‘Zed’ que significa “*Zynq Evaluation and Development*” y está constituida por Xilinx, Avnet (distribuidor) y Digilent (fabricante). La *ZedBoard* destaca entre los estudiantes debido a su precio accesible (\$ 449.00 US Dlls) y porque cuenta con una comunidad en línea para los usuarios de esta tarjeta. Más

adelante se presenta la arquitectura y características que ofrece esta tarjeta constituida por un SoC *Zynq* 7000 y las herramientas de diseño de Xilinx para la *ZedBoard*.

La *ZedBoard* cuenta con un dispositivo XC72020 *Zynq* de los más pequeños en la familia *Zynq* 7000 y se basa en un FPGA de la familia Artix-7 con una capacidad de 13,300 *slices*, 229 DSP48E1 (bloques de DSP) y 140 BlockRAM. El dispositivo también contiene un bloque IP XADC [42].

Las características de la *ZedBoard* son:

- Memoria:
  - Memoria RAM 512 MB DDR3 (módulos de 128 Mb x 2).
  - 256 Mb de memoria QSPI *Flash*.
- Interfaces Entrada/Salida
  - USB-JTAG.
  - Interfaz de red de 10/100/1G *Ethernet*.
  - USB OTG 2.0.
  - Lector de tarjetas SD.
  - USB 2.0 FS USB-UART.
  - 5 conectores PMOD (1 al PS, 4 al PL).
  - 1 conector LPC FMC.
  - 1 AMS *Header*.
  - 2 botones de *reset* (1 para PS, 1 para PL).
  - 7 botones (2 PS, 5PL).
  - 8 interruptores conectados únicamente al PL.
  - 9 LED para el usuario (1 PS, 8 PL).
- Relojes en placa:
  - 33.33 MHz para el PS.
  - 100 MHz dirigidos al PL.
- Vídeo/Audio:
  - Salida HDMI.
  - Salida VGA de 12-bit (RGB 444).
  - 128x32 OLED *Display*.
  - Códec de audio Analog Devices ADAU1761, entrada de micrófono y salida para auriculares

El diagrama de bloques de la *ZedBoard* se presenta en la Figura 2.8. Se muestra cada uno de los componentes de la tarjeta, así como la capacidad de bits en los buses de comunicación hacia cada uno de los componentes.

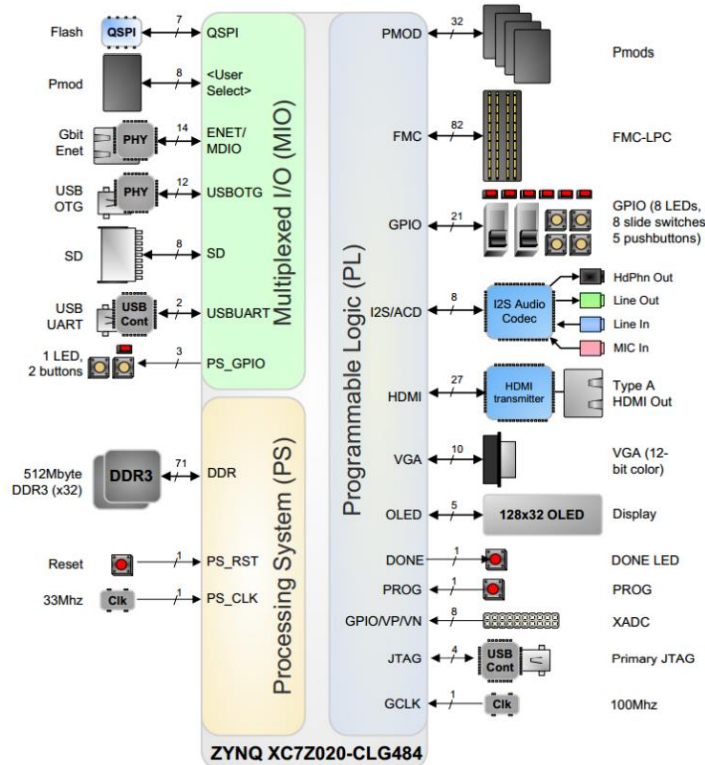


Figura 2.8 Diagrama de bloques de la tarjeta *ZedBoard* [42].

## 2.5. Cámara de video OV7670

La empresa *OmniVision Technologic* es una empresa encargada de desarrollar y ofrecer soluciones avanzadas de imágenes en una gran variedad de mercados industriales y de consumo. *OmniVision* ofrece el sensor CMOS de imagen OV7670 de bajo voltaje, con distintas funciones de control de imagen, tales como control de exposición automática (AEC, por sus siglas en inglés), control del balance de blancos (AWB, por sus siglas en inglés), cancelación de ruido, control automático o manual de brillo, saturación y gamma, entre otros parámetros. Además, cuenta con la posibilidad de trabajar a distintas resoluciones, formatos (VGA, QVGA, CIF, QCIF) y tasa de imagen de hasta un máximo de 640 x 480 píxeles a 30 cuadros por segundo (FPS, por sus siglas en inglés) y en diferentes espacios de color RGB555 (5 bits de intensidad para cada componente de color: R (*Red*), G (*Green*), B (*Blue*)), RGB565 (5 bits de información

para R, B y 6 bits para G) y YUV422 (4 bits de información para Y (*Intensity*), y 2 bits de información para U (*Hue*) y V(*Value*)).

Todas estas funcionalidades de procesamiento pueden ser programadas mediante la interfaz SCCB (*Serial Camera Control Bus*) desarrollada por *OmniVision*. La interfaz se describirá en el capítulo 3 con más detalle y se mostrará su parecido al protocolo I2C [43]. El esquema general de las componentes de la cámara OV7670 se muestra en la Figura 2.9.

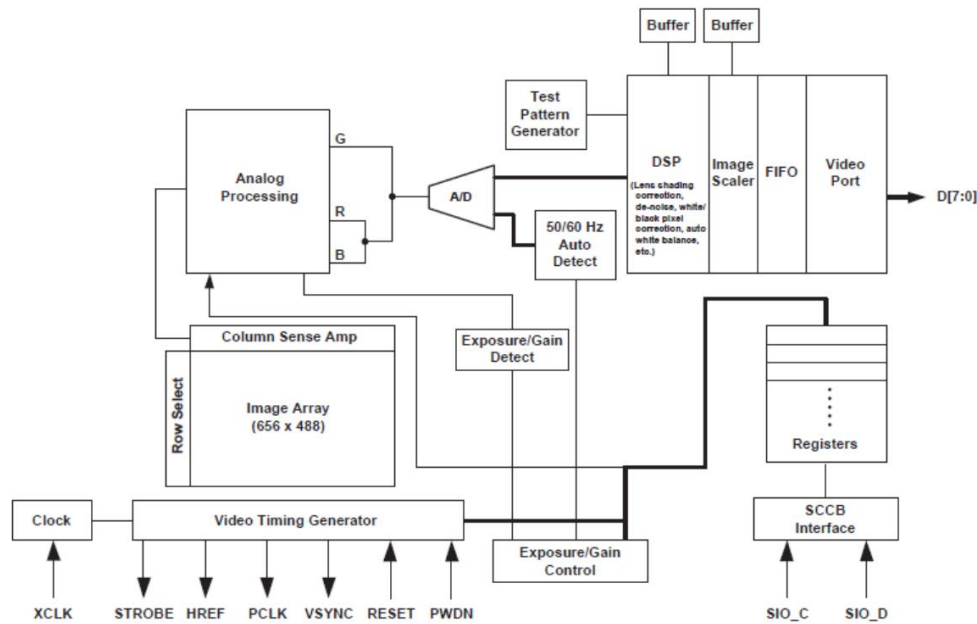


Figura 2.9 Esquema de la cámara OV7670 [43].

La función de algunas de las componentes es:

- **Arreglo de Sensores de Imagen (*Image Sensor Array*):** sensor de 656 x 488 pixeles para un total de 320.128 pixeles, de los cuales 640 x 480 son activos (307.200).
- **Generador sincronizador de video (*Video Timing Generator*):** es el componente encargado de realizar funciones como la generación y distribución de señales de reloj, tanto internas como externas (VSYNC, HREF, PCLK), control del sensor de la imagen, generación de imagen y control de tasa de imágenes por segundo.

- Procesador de señal analógica (*Analog Signal Processor*): se encarga de realizar todas las funcionalidades analógicas de la imagen incluyendo el control automático de ganancia y el balance de blancos. El bloque separa la señal en las componentes RGB.
- Convertidor Análogo/Digital (*A/D Converter*): se encarga de convertir la señal analógica de entrada a una señal digital de 10 bits. Este convertidor trabaja a una frecuencia máxima de 12 MHz.
- Generador de patrones de prueba (*Test Pattern Generator*): encargado de generar patrones de color y de escala de grises de 8 bits como prueba del sistema.
- Procesador de señales digitales (*DSP, Digital Signal Processor*): controla la señal original en RGB incorporando un control de calidad de imagen con funciones como: realce de bodes, convertidor de espacio de color (RGB o YUV), control de saturación, corrección de pixeles y de ruido, control automático de gamma y convertidor de 10 a 8 bits.
- Escalador de imagen (*Image Scaler*): controlador encargado de adaptar los datos de salida al formato previamente configurado. Puede escalar la imagen de VGA a CIF.
- Interfaz SCCB (*SCCB Interface*): interfaz que controla toda la configuración del sensor OV7670.

La hoja de datos del sensor [43] describe la funcionalidad del sensor y de los 18 pines con que cuenta, la Tabla 2.2 muestra el nombre, tipo y funcionalidad de cada pin de conexión.

Tabla 2.2 Pines del sensor OV7670.

| Nombre del pin | Tipo           | Función                             |
|----------------|----------------|-------------------------------------|
| SIOD           | Entrada/Salida | Datos interfaz SCCB                 |
| SIOC           | Entrada        | Reloj de sincronización SCCB        |
| RESET          | Entrada        | Reinicia sensor en nivel bajo       |
| PWDN           | Entrada        | Apaga sensor en nivel alto          |
| HREF           | Salida         | Señal de referencia horizontal      |
| VSYNC          | Salida         | Señal de referencia vertical        |
| XCLK           | Entrada        | Reloj del sistema                   |
| PCLK           | Salida         | Reloj de sincronización de pixel    |
| D0             | Salida         | Bit 0 del valor del pixel de salida |
| D1             | Salida         | Bit 1 del valor del pixel de salida |

|     |                         |                                     |
|-----|-------------------------|-------------------------------------|
| D2  | Salida                  | Bit 2 del valor del pixel de salida |
| D3  | Salida                  | Bit 3 del valor del pixel de salida |
| D4  | Salida                  | Bit 4 del valor del pixel de salida |
| D5  | Salida                  | Bit 5 del valor del pixel de salida |
| D6  | Salida                  | Bit 6 del valor del pixel de salida |
| D7  | Salida                  | Bit 7 del valor del pixel de salida |
| VCC | Voltaje de alimentación | Alimentación del sensor a 3.3V      |
| GND | Tierra                  | Tierra                              |

Un punto importante que considerar son las señales de sincronía (HREF, VSYNC, PCLK) y se describirá su funcionalidad para dar paso al diseño de la comunicación con el sistema.

**HREF:** la señal de referencia horizontal, indica cuando termina la línea de pixeles horizontal en nivel bajo, por lo tanto, siempre envía datos cuando se encuentra en nivel alto.

**VSYNC:** señal de referencia vertical, indica cuando termina de enviarse una imagen completa o un cuadro y comienza a transmitirse el siguiente cuadro. En nivel alto indica cuando el cuadro terminó de transmitirse y en nivel bajo cuando comienza a transmitirse el siguiente cuadro.

**PCLK:** señal de sincronía del pixel, la señal de reloj en alto indica cuando leer el pixel está listo para leerse por los pines de datos de salida.

Los mapas de tiempo de las señales HREF, VSYNC antes descritas se muestran en la Figura 2.10 y Figura 2.11.

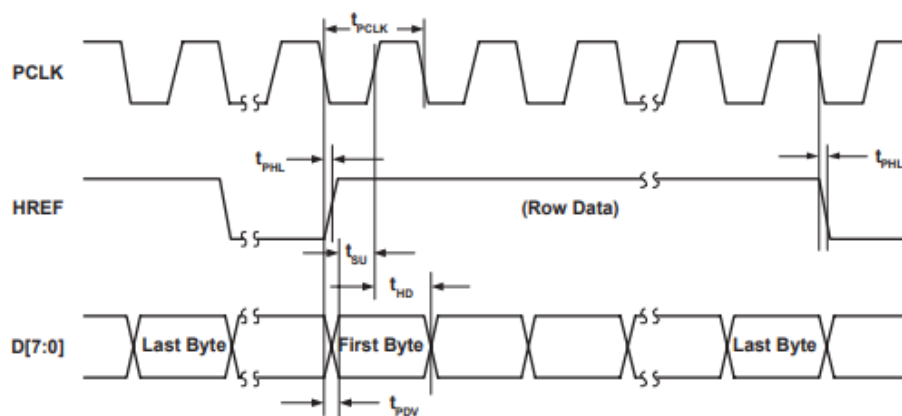


Figura 2.10 Señal de sincronización horizontal [43].

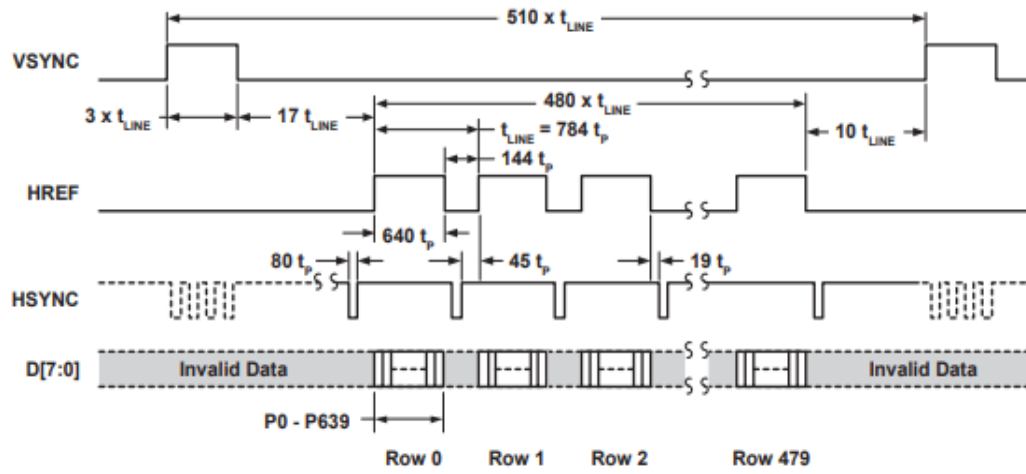


Figura 2.11 Señal de sincronización vertical [43].

Para mostrar el mapa de tiempos de la señal PCLK obteniendo un mejor contexto, se explicará el espacio de color de salida utilizado en esta tesis, el cual es el RGB565, uno de los espacios disponibles en el sensor OV7670. La información de color queda almacenada en 16 bits correspondientes a 5 bits, 6 bits y 5 bits para cada uno de los colores rojo, verde y azul respectivamente. En la transmisión de los valores de color, se emplean dos ciclos de reloj PCLK, en cada ciclo de PCLK se envía un byte de información, si se sincroniza con HREF, el primer byte contiene 5 bits de información correspondiente al color rojo y 3 bits de información del color verde, correspondientes a los menos significativos, en el segundo ciclo PCLK se obtiene la información faltante del color verde la cual corresponde a 3 bits, y por último se tienen los 5 bits de color azul, como se muestra en la Figura 2.12. Cabe mencionar que los formatos RGB555 y YUV422, son similares en la transmisión de datos antes descrita, esto quiere decir que también se transmiten en paquetes de 2 bytes [43].

En la sección 2.4 se menciona que la tarjeta *ZedBoard* tiene una salida VGA de 12 bits en el espacio RGB, donde los 12 bits se distribuyen equitativamente (4 bits por color), es decir, RGB444, siendo una característica de la tarjeta *ZedBoard*. Por lo tanto, se tendrá en cuenta esta información para diseñar el módulo encargado de tomar datos del sensor OV7670 y el despliegue de imagen por VGA [42].

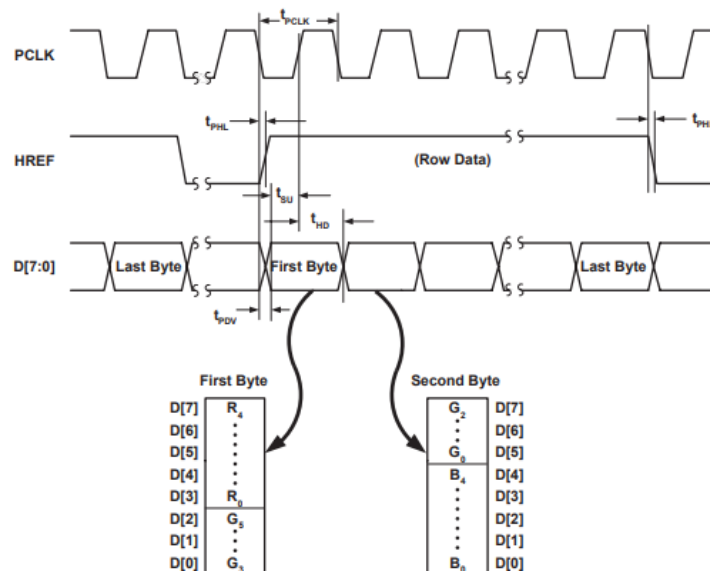


Figura 2.12 Mapa de tiempos PCLK y transmisión de datos RGB565 [43].

En el presente capítulo se describió el hardware utilizado en el desarrollo del proyecto de tesis y detalles de los elementos importantes a tomar en cuenta para el diseño del sistema final al que se llegó. Las arquitecturas del hardware estudiadas son de alta complejidad: el SoC-FPGA *Zynq 7000*, la tarjeta *ZedBoard*, el sensor de imagen *OV7670* y los periféricos asociados. Se complementa el hardware con los elementos de software y lenguajes descriptivos de hardware para desarrollo de los algoritmos implementados los cuales se presentan en los capítulos siguientes.

## CAPÍTULO III

### ALGORITMOS PARA PROCESAMIENTO DIGITAL DE IMÁGENES

Como se observó en capítulos anteriores, los sistemas de visión por computadora hacen uso de algoritmos para la extracción de características que ayuden a dar solución a problemas en específico. Los algoritmos que se usaron para el desarrollo de este trabajo están ampliamente descritos en la literatura relacionada con el tema y en este capítulo se describen los aspectos más importantes de los mismos. Los algoritmos que se definen se eligieron por razones de costo en tiempo y recursos de diseño en la plataforma *ZedBoard*.

#### 3.1. Tipos de procesamiento de imágenes digitales

En este proyecto se usaron algoritmos de procesamiento de imágenes pixel por pixel o punto a punto, de ventana, y geométrico [44].

El **procesamiento punto a punto** se representa por una operación de transformación a cada uno de los píxeles que conforman una imagen  $I$  de entrada, tal que:

$$I'(x, y) = T[I(x, y)] \quad (3.1)$$

donde  $I(x, y)$  es el pixel original antes de la transformación por medio del operador  $T[]$ , y como resultado se obtiene el pixel resultante denotado como  $I'(x, y)$ .

El **procesamiento de ventana** se denota por una operación de transformación al conjunto de vecinos más cercanos al pixel de entrada a procesar, de tal manera que:

$$I'(x, y) = T[I_8(x, y)] \quad (3.2)$$

donde  $I_8(x, y)$  son los píxeles vecinos del pixel bajo transformación.

Para finalizar, en los procesamientos anteriores, el pixel nuevo queda en la misma posición  $(x, y)$  original; en el **procesamiento geométrico** se genera un nuevo pixel, pero no necesariamente en la misma posición del o de los píxeles utilizados en el cálculo, tal que,

$$I'(s, t) = T[I_m(x, y)] \quad (3.3)$$

donde  $I_m$  es el pixel bajo la transformación geométrica.

### 3.2. Transformación RGB a escala de Grises

Las imágenes digitales en sistemas de visión son captadas por sensores, los sensores comunes ofrecen un resultado en un formato en color tridimensional (3D) RGB. La mayoría de los algoritmos recurre a una reducción de dimensiones en el color para aminorar el tiempo de procesamiento. Para disminuir la dimensión 3D de la imagen se utiliza una transformación a escala de grises (1D) y así se trabaja solo en una dimensión. En este proyecto se analizaron algoritmos con imágenes en escala de grises, para esto se utilizó una función integrada en la herramienta HLS de la cual se discute ampliamente en el capítulo 5. La función consta de un porcentaje de cada color (R, G, B) en la imagen resultante  $g(x, y)$  en niveles de gris [44],

$$g(x, y) = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \quad (3.4)$$

Cuando tiene la imagen en escala de gris, esta se puede modificar para poder obtener imágenes distintas, ya sea el negativo de la imagen o binarización. Para poder obtener el negativo,  $I_N(x, y)$ , de la imagen  $g(x, y)$  en un rango de intensidad  $[0, L-1]$ , se obtiene por medio de (3.5) [44]:

$$I_N(x, y) = L - 1 - g(x, y) \quad (3.5)$$

Cabe mencionar que la ecuación (3.5) se puede aplicar a cada canal de RGB obteniendo el negativo de una imagen a color.

Para la binarización de una imagen, la cual nos puede servir como un discriminante debido a que podemos resaltar solo las intensidades que presentan objetos que son de interés. La expresión queda en función de un umbral  $T$  el cual indica el mapeo de los pixeles originales al resultante, tal que:

$$I_B(x, y) = \begin{cases} 0 & \text{para } I(x, y) \leq T \\ 1 & \text{para } I(x, y) > T \end{cases} \quad (3.6)$$

se observa que el mapeo es a un valor binario y donde el umbral  $T$  toma un valor entre el rango de la intensidad máxima y mínima.

### 3.3. Detección de bordes Sobel

Para la detección de bordes en una imagen digital se tienen métodos tradicionales como son los de gradiente: Roberts, Prewitt y Sobel. En estos métodos se define que dos puntos discretos de una imagen son una función continua de dos dimensiones [44]. Así, la función es aproximada a partir de dos puntos y se obtiene el gradiente de esta función como un indicador de que tanto cambia la intensidad de una imagen en cada punto, estos métodos se basan en el máximo y el mínimo de la primera derivada de la imagen.

El método seleccionado en este proyecto fue el operador Sobel, donde la aproximación se hace con base a nueve puntos que forman una ventana de  $3 \times 3$  tomando los ocho vecinos más cercanos de cada pixel a procesar. Las matrices en  $g_x$  y  $g_y$ , son los gradientes que se usan para encontrar los cambios de intensidad en la imagen  $I$ , las ventanas de derivadas  $g_x$  y  $g_y$  se usan para la convolución ( $\otimes$ ) con la imagen original, tal que,

$$g_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad g_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad (3.7)$$

$$I_x = g_x \otimes I \quad I_y = g_y \otimes I \quad (3.8)$$

La magnitud,  $|G|$ , y la dirección,  $\theta$ , de los gradientes se calcula con:

$$|G| = \sqrt{(g_x \otimes I_x)^2 + (g_y \otimes I_y)^2} \quad (3.9)$$

$$\theta \approx \tan^{-1} \left( \frac{I_x}{I_y} \right) \quad (3.10)$$

### 3.4. Detección de esquinas por Harris

El algoritmo para detección de esquinas en imágenes por Harris y Stephens se genera en 1988 [45] y se basa en un trabajo previo de Moravec H. de 1980 [46]. Este algoritmo, para la detección

de las esquinas hace uso de la diferencia en intensidad de los gradientes de una imagen en dirección horizontal y vertical. Un cambio de intensidad en ambas direcciones en el mismo vecindario indica un borde y ningún cambio significativo indica una zona plana. Harris y Stephens utilizan gradientes y valores propios para detectar cambios de intensidad en todas las direcciones [45], [47]. Para comprender la detección de equinas por este método, se considera una imagen en escala de grises,  $I$ , a la cual se le extraen los gradientes en dirección horizontal y vertical,  $I_x$  e  $I_y$ , respectivamente. Después se llevan a cabo las operaciones de convolución  $(I_x)^2$ ,  $(I_y)^2$  y  $(I_x I_y)$  con una ventana Gaussiana,  $w$ , para obtener la matriz de autocorrelación  $M$ . En las siguientes ecuaciones se formaliza la metodología previamente descrita:

Obtención de gradientes usando el método de Sobel:

$$I_x = g_x \otimes I, \quad I_y = g_y \otimes I \quad (3.11)$$

convoluciones con la ventana Gaussiana  $w$

$$\bar{I}_x^2 = I_x^2 \otimes w, \quad \bar{I}_y^2 = I_y^2 \otimes w, \quad \bar{I}_{xy} = I_x I_y \otimes w \quad (3.12)$$

tal que la matriz de autocorrelación  $M$  se calcula:

$$M = \begin{bmatrix} \bar{I}_x^2 & \bar{I}_{xy} \\ \bar{I}_{xy} & \bar{I}_y^2 \end{bmatrix} \quad (3.13)$$

Se analizan los eigenvalores de la matriz  $M$  para expresar la caracterización de las esquinas, por lo tanto,  $M$  tiene eigenvalores  $(\lambda_1, \lambda_2)$  “altos” para la detección en puntos de interés, esto quiere decir:

- Si  $\lambda_1 \approx 0$  y  $\lambda_2 \approx 0$  entonces el pixel  $(x, y)$  analizado no tiene variaciones de intensidad con sus vecinos indicando una sección plana.
- Si  $\lambda_1 \gg \lambda_2$  o  $\lambda_1 \ll \lambda_2$  indican la existencia de una borde en la ventana analizada.
- Si  $\lambda_1$  y  $\lambda_2$  son valores elevados indican una esquina.

El mapa de respuesta de los eigenvalores se expresa en la Figura 3.1 clasificando las respuestas antes mencionadas. Los valores de esquinas  $C(x,y)$ , se calcula por medio del determinante ( $Det$ ) y la suma de la diagonal principal de la matriz  $M$  ( $trace$ ):

$$Det(M) = \lambda_1 \cdot \lambda_2 = (\bar{I}_x^2 \cdot \bar{I}_y^2) - (\bar{I}_{xy})^2 \quad (3.14)$$

$$trace(M) = \lambda_1 + \lambda_2 = \bar{I}_x^2 + \bar{I}_y^2 \quad (3.15)$$

donde  $\lambda_1$  y  $\lambda_2$  son los eigenvalores de  $M$  y por lo tanto, los valores usados en la detección de esquinas:

$$C(x,y) = Det(M) - k \cdot Trace^2(M) = \lambda_1 \lambda_2 = (\bar{I}_x^2 \bar{I}_y^2) - (\bar{I}_{xy})^2 \quad (3.15)$$

El parámetro  $k$  en la ecuación (3.15) es ajustable y determinado empíricamente, en las paqueterías de software donde se utiliza este método suelen utilizarse valores de  $k$  entre 0.04 y 0.15.

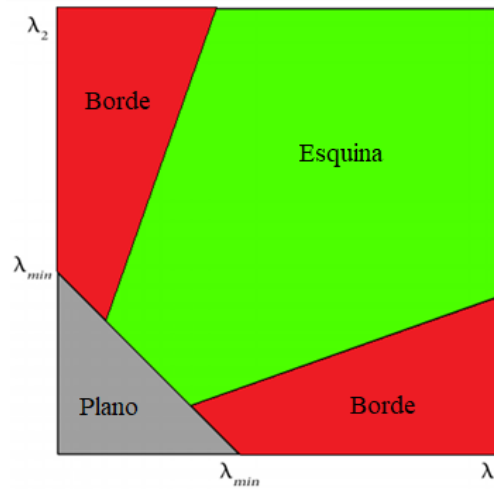


Figura 3.1 Representación del método Harris en el espacio  $\lambda_1$ - $\lambda_2$ .

### 3.5. Transformada de Hough

La transformada de Hough (TH) se usa como un algoritmo para la detección de formas, obteniendo una cantidad infinita de formas que pueden pasar por cada pixel en la imagen. Sin embargo, cada una de ellas debe satisfacer la ecuación de la forma a buscar en la imagen [44], [48]. La TH utiliza un acumulador discreto con una dimensión para cada grado libertad. Cada punto en el acumulador corresponde a una forma con valores de parámetros específicos, discretizados a una resolución elegida apropiadamente. El valor de un punto en el acumulador es el número de pixeles que cumple esa parametrización específica, lo que facilita la clasificación de la parametrización más destacado en una línea. La complejidad computacional de la TH es lineal con respecto al número de pixeles considerados, ya que cada pixel genera una cantidad fija de candidatos de parametrización.

La detección de la forma más común en la que se utiliza la TH es para la detección de líneas en una imagen. Las líneas se pueden encontrar en las imágenes, representando todas las posibles rectas que puedan aparecer en una imagen. Una línea se puede representar con la ecuación de la línea recta como  $y = mx + c$ , o en forma paramétrica como  $\rho = x(\cos \theta) + y(\sin \theta)$ , donde  $\rho$  es la distancia perpendicular desde el origen a la recta, y  $\theta$  es el ángulo formado entre esta perpendicular a la recta y el eje horizontal como se observa en la Figura 3.2a. Usando esta parametrización, cada punto en el acumulador corresponde a una línea en el espacio  $(x,y)$ , y cada punto en la imagen corresponde a una curva sinusoidal en el acumulador  $(\rho,\theta)$ , Figura 3.2b, donde el valor más alto en el acumulador representa la recta con más pixeles en común de la imagen de entrada. Posteriormente los datos candidatos a línea pasan a ser procesados y ser mapeados nuevamente al espacio  $(x,y)$ .

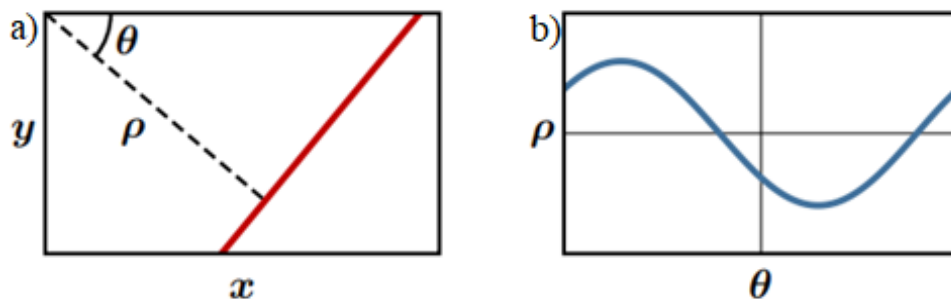


Figura 3.2(a) Espacio  $x$ - $y$ , y (b) espacio de Hough  $\rho$ - $\theta$  [49].

Uno de los inconvenientes en la TH es el número infinito de líneas que pueden representarse para un punto, por lo que lo hace costoso computacionalmente y debe de elegirse una resolución apropiada en la implementación para cada parámetro dependiendo de la aplicación.

Se han descrito los algoritmos implementados en el sistema de procesamiento de imágenes en base al SoC-FPGA. Es importante mencionar que los algoritmos punto a punto y de ventana (Sobel) pueden modificarse para implementar otra clase de procesamiento de manera sencilla al modificar los valores de la ventana (coeficientes). Los algoritmos de detección de esquinas y Hough los consideramos de mediana y alta complejidad, respectivamente y de gran utilidad en diversas aplicaciones.

## CAPÍTULO IV

### DESARROLLO DEL SISTEMA EN SoC-FPGA

En este capítulo se describe el desarrollo de la elaboración del sistema de captura de imagen, guardado de imagen en memoria y despliegue de imagen por el periférico de visualización. Se describen los módulos realizados y procesamiento básicos en la imagen capturada, los cuales van desde inversión de colores a una binarización de imagen. Se retomarán conceptos vistos en el capítulo 2 para el diseño del sistema de captura de imagen y desplegado

#### 4.1. Diseño de módulo de configuración del sensor en VHDL

En el capítulo 2 se describió el sensor OV7670, por lo que se trabajó bajo ese concepto para la generación de un modo que capte los píxeles en formato RGB565, siendo necesario el diseño del módulo en el *Zynq* encargado de configuración y gestión de la cámara.

El módulo tiene la funcionalidad de controlar la configuración de la cámara mediante el protocolo de comunicación SCCB (*Serial Camera Control Bus*), diseñado por OmniVision para sus dispositivos (con cierto parecido a I2C) [50]. El sensor utiliza una versión simplificada de 2 hilos debido a que la placa en la que se encuentra tiene acceso a menos pines de los que realmente se disponen en el chip. Este bus se basa en el modelo maestro/esclavo (*master/slave*), muy similar al bus I2C, en el que un dispositivo maestro se conecta al bus SCCB para controlar al menos un dispositivo esclavo. El esquema de comunicación se muestra en la Figura 4.1, donde:

- SIOC: hilo controlado por el maestro. A nivel alto constante el bus está libre y a nivel bajo constante el sistema está en reposo. Cuando el bus está activo va cambiando de nivel alto a bajo, actuando como el reloj que sincroniza la transmisión.
- SIOD: pueden acceder al hilo tanto el dispositivo maestro como el esclavo. Es la conexión que transporta los bits de datos en la transmisión. Cuando el bus está libre queda a nivel alto. A nivel bajo cuando el sistema se suspende.



Figura 4.1 Esquema del protocolo SCCB

El sensor solo permite que un maestro lo controle, por lo tanto, la tarjeta de desarrollo *ZedBoard* (maestro) será el dispositivo encargado de configurar el sensor (esclavo).

En la implementación OmniVision advierte de que el dispositivo maestro debe ser capaz de poner la línea de datos, SIOD, en tres estados: alto ('1'), bajo ('0') y alta impedancia ('Z'). La tarjeta *ZedBoard* puede soportar los tres estados, por lo que no hay problema a la hora de usar el bus.

#### 4.1.1. Etapas de transmisión de datos para la configuración del sensor OV7670

Una vez explicado el cómo se trabaja con la comunicación SCCB, se comenzará con el uso de las señales SIOC y SIOD para la comunicación. La señal SIOC muestra cuando se transmite cada valor binario con señales en alto. El valor binario transmitido, se indica con la señal SIOD, teniendo en cuenta que el valor debe estar presente durante un periodo de SIOC. Se puede observar que la señal SIOD funge como E/S, por lo tanto, será una señal “*INOUT*” en VHDL.

Para el diseño del módulo de configuración, se comenzó con el análisis del diagrama de tiempo para la transmisión de datos en la Figura 4.2.

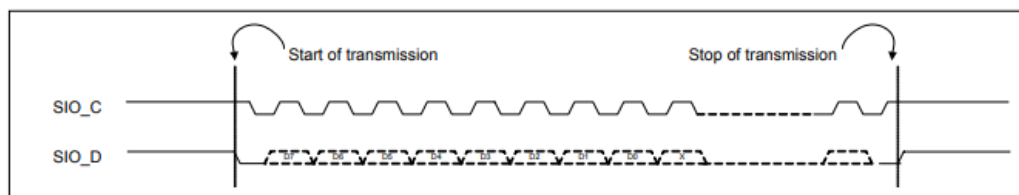
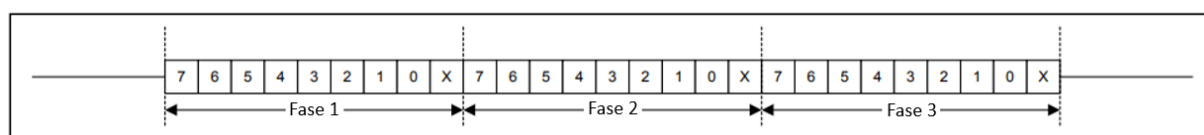


Figura 4.2 Diagrama de tiempos para la transmisión de datos mediante protocolo SCCB [50].

En el estado de reposo posterior a la transmisión de datos, SIOC se encuentra en alto ‘1’, y SIOD en alta impedancia ‘z’. Cuando SIOD pasa de ‘z’ a ‘0’ o bajo, se indica al sensor que inicia la transmisión de datos mientras que SIOC se mantiene en ‘1’, como se muestra en la Figura 4.2. Cuando ambos SIOC y SIOD están en ‘0’, inmediatamente comienza la transmisión de datos, inicia con el primer bit en el flanco ascendente más cercano de SIOC. Para detener la transmisión, una vez transmitido el último bit, SIOC y SIOD pasan a nivel alto (Figura 4.2).

La comunicación con el dispositivo esclavo se divide en tres fases de transmisión de ocho bits: dirección de identificación del dispositivo (*ID Address*), dirección del registro, dato a escribir. En cada ciclo de transmisión completo se escribe un byte de datos en un registro del dispositivo esclavo, de acuerdo con el esquema de la Figura 4.3. Los bits se transmiten en serie, siendo el más significativo el primero que se envía en cada etapa o fase de transmisión.



Fase 1: Identificación del dispositivo - Fase 2: Dirección del registro - Fase 3: Dato a escribir

Figura 4.3 Esquema de transmisión de un byte [27].

La identificación consta de ocho bits e identifica al dispositivo al que el maestro intenta acceder, en este caso, la dirección del sensor OV7670 para escritura es “0x42” (esta dirección es exclusiva para cada dispositivo de OmniVision, se debe consultar en su hoja de datos). La dirección de cada registro viene especificada en la documentación del sensor, abarcando de “0x00” a “0xC9”, aunque especifica que hay más registros, pero son reservados. Cada registro cuenta con 8 bits, cuya función se consulta en la documentación para conseguir la configuración deseada. Para separar cada fase se transmite un noveno bit, llamado por el fabricante “*Don’t-care bit*”. Este es un bit de relleno en el que SIOD toma el estado de alta impedancia, razón por la cual es necesario que el dispositivo maestro pueda manejar los tres estados antes mencionados, toda la información detallada sobre SCCB para dispositivos de OmniVision se encuentra en [50].

#### 4.1.2. Diseño VHDL para la configuración del sensor OV7670

Para el diseño del módulo encargado de configurar el sensor, se tomó en cuenta la información del protocolo antes detallado: etapas de inicio, paro y transmisión de datos. Se propuso un diseño por jerarquía el cual contiene submódulos, siendo *ov7670\_contoller.vhd* el módulo principal para la configuración de la cámara. En *ov7670\_contoller.vhd* se instancian submódulos encargados de la comunicación SCCB y manejo de registros *i2c\_sender.vhd* (nombre dado por el parecido entre la comunicación I2C y SCCB) y *ov7670\_registers.vhd* respectivamente. El desarrollo de los submódulos se basó en la página HAMSTERWORKS [51], con créditos al Ing. y usuario de internet Mike Field ‘Hamster’ por su trabajo y dedicación por la información didáctica y detallada de la interfaz SCCB y VGA.

A continuación, se describirán cada uno de los submódulos, así como el módulo encargado de controlar el proceso de configuración:

**Submódulo *i2c\_sender.vhd*:** cuenta con la entrada *send* para indicar cuando se debe de iniciar un ciclo de transmisión. Cuando se activa la entrada *send* se cargan los 24 bits a enviar (*id*, *reg*, *value*) la señal interna *data\_sr* y comienza el proceso de transmisión antes descrito. Una vez terminada la transmisión se activa la salida *taken*, indicando al módulo el cargar los siguientes 24 bits a enviar, una vez cargados los datos comienza el ciclo de transmisión nuevamente, y *taken* pasa a ‘0’. La asignación de los datos se muestra a continuación sobre la señal interna *data\_sr*:

```
data_sr <= "100" & id & '0' & reg & '0' & value & '0' & "01";
```

se puede observar cómo se cargan los 24 bits necesarios y a cada uno de los bytes se le agrega el bit “*Don’t-care bit*”. En cada ciclo se cargan los tres bytes en la señal *data\_sr*, a la que se van añadiendo ceros a la derecha cada vez que se manda un bit, dejando su bit más significativo conectado a la salida SIOD, de forma que se van enviando los 24 bits de uno en uno. La entidad de este módulo se muestra en las siguientes líneas descriptivas de hardware:

```
entity i2c_sender is
  Port ( clk      : in   STD_LOGIC;
        siod     : inout STD_LOGIC;
        sioc     : out   STD_LOGIC;
        taken    : out   STD_LOGIC;
        send     : in   STD_LOGIC;
```

```

        id      : in  STD_LOGIC_VECTOR (7 downto 0);
        reg     : in  STD_LOGIC_VECTOR (7 downto 0);
        value   : in  STD_LOGIC_VECTOR (7 downto 0));
end i2c_sender;

```

**Submódulo `ov7670_registers.vhd`:** en este submódulo se encuentran los valores de cada registro a configurar, este mismo módulo se encarga de pasar los valores de los registros a `i2c_sender.vhd` para que sean enviados a la cámara. El submódulo rota entre los valores de los registros a enviar cuando un ciclo de transmisión se ha completado. La entidad de este módulo se muestra a continuación,

```

entity ov7670_registers is
    Port ( clk      : in  STD_LOGIC;
          resend    : in  STD_LOGIC;
          advance   : in  STD_LOGIC;
          command   : out  std_logic_vector(15 downto 0);
          finished  : out  STD_LOGIC);
end ov7670_registers;

```

La señal *advance* esta instanciada a la señal *taken* la cual indica el momento en el que se debe enviar el siguiente registro, la señal *resend* esta agregada por si se desea volver a configurar el sensor. Para el envío de los registros se utiliza la dirección (*address*, señal interna) en una sentencia *case*, la señal se incrementa de uno en uno cada vez que la señal *advance* se activa y se envían dos bytes de información (*reg* y *value*) por medio de la señal de salida *command*. Por último, se debe mencionar que los submódulos se diseñaron para operar con la misma frecuencia de operación, esto para sincronización. Como paso final, para detener el envío de datos, se implementó una salida de un bit (*finished*) que se activa cuando la configuración de los registros ha sido enviada y el sensor está completamente configurado. Como indicador en tiempo real, esta señal quedará conectada a un LED de la placa para que éste quede encendido cuando la configuración haya terminado.

**Módulo `ov7670_contoller.vhd`:** como el diseño está basado en niveles jerárquicos, este módulo se encuentra en el top de los dos antes mencionados, en él se instancian los submódulos para dar un control al proceso de configuración del sensor. En esta etapa de instanciación, se puede utilizar el diagrama a bloques que proporciona Vivado o la sentencia `COMPONENT` en VHDL para su instanciación, en este módulo se usó la sentencia `COMPONENT`. En este módulo

se manejan las señales de control del sensor: SIOD, SIOC, XCLK, PWDN y RESET. Las señales SIOD, SIOC ya se describieron.

La señal de reloj XCLK, es la encargada de sincronizar la captura de la imagen, y la señal de reloj con la que se trabaja es de 50 MHz. PWDN es la señal encargada de poner el sensor en modo reposo, este modo se activa cuando PWDN está en '1', en la descripción de hardware diseñada siempre está en '1'. La señal *RESET* esta instanciada a la señal *resend*, la cual reinicia los registros del sensor.

Como módulo final, el diseño se puede observar en la Figura 4.4 en forma de IP para su utilización en el sistema de procesamiento de imágenes, donde el reloj de entrada es de 50 MHz.

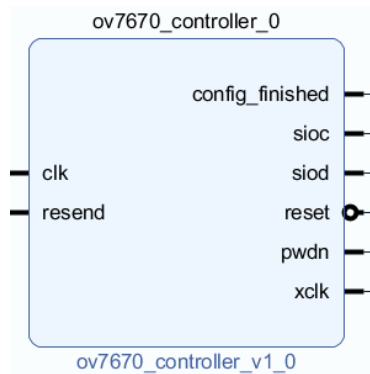


Figura 4.4 IP módulo encargado de configurar el sensor.

La entidad del diseño final para la configuración quedó de la siguiente forma:

```
entity ov7670_controller is
    Port ( clk      : in    STD_LOGIC;
          resend   : in    STD_LOGIC;
          config_finished : out std_logic;
          sioc     : out    STD_LOGIC;
          siod     : inout  STD_LOGIC;
          reset    : out    STD_LOGIC;
          pwn      : out    STD_LOGIC;
          xclk     : out    STD_LOGIC
    );
end ov7670_controller;
```

## 4.2. Módulo de captura de imagen

Esta etapa de desarrollo del sensor de video incluye el módulo, *ov7670\_capture.vhd*, encargado de la captura de datos de la imagen pixel por pixel en formato RGB5:6:5. Siguiendo con la información descrita en el capítulo 2 del formato y diagramas de tiempo para la información adquirida del sensor (píxeles de la imagen). Se comenzará con las señales fundamentales de sincronía del sensor para este módulo: PCLK, HREF y VSYNC.

La señal PCLK indica cuando es el tiempo de lectura de cada byte de información, por lo tanto, será la frecuencia de operación del módulo. Cuando las señales HREF y VSYNC están en '1' le indican al sensor que envíe los datos válidos, por lo que se infiere en el diseño el inicio en la trama de datos válidos solo cuando PCLK, HREF estén en alto y VSYNC estén en bajo. Para la lectura de los datos correctamente es necesaria la sincronización con la señal PCLK y recibir en el primer ciclo el byte correspondiente a los cinco bits de color rojo y los tres bits más significativos de color verde, y en el segundo ciclo de PCLK se reciben los tres bits menos significativos del color verde y los cinco bits del color azul. En cada dos ciclos de reloj de PCLK se obtiene un pixel de la imagen codificada en 12 bits, tal como se muestra en la Figura 2.10.

El siguiente problema es el guardado del pixel en sus 12 bits. Vivado proporciona una herramienta capaz de generar una memoria para el guardado de los datos, el *Block Memory Generator*. El generador de memoria brinda un bloque RAM que se puede instanciar mediante VHDL o por integración de diseño de bloques, y se puede configurar dependiendo de la aplicación, la información detallada se encuentra [52]. El bloque de memoria RAM proporcionado hace posible generar una arquitectura *Simple Dual Port RAM* que es esencialmente una memoria con puerto de escritura y puerto de lectura.

Para tener acceso a esta memoria RAM es necesario proporcionar los parámetros de RAM del bloque a instanciar como se muestra en la Figura 4.5. Una vez abierta la ventana de configuración, se escoge el tipo de memoria como *Simple Dual Port RAM*, un ancho para datos de 12 bits y un total de 307,200 datos (480\*640), tanto para el puerto A como para el puerto B como se muestra en la Figura 4.6.

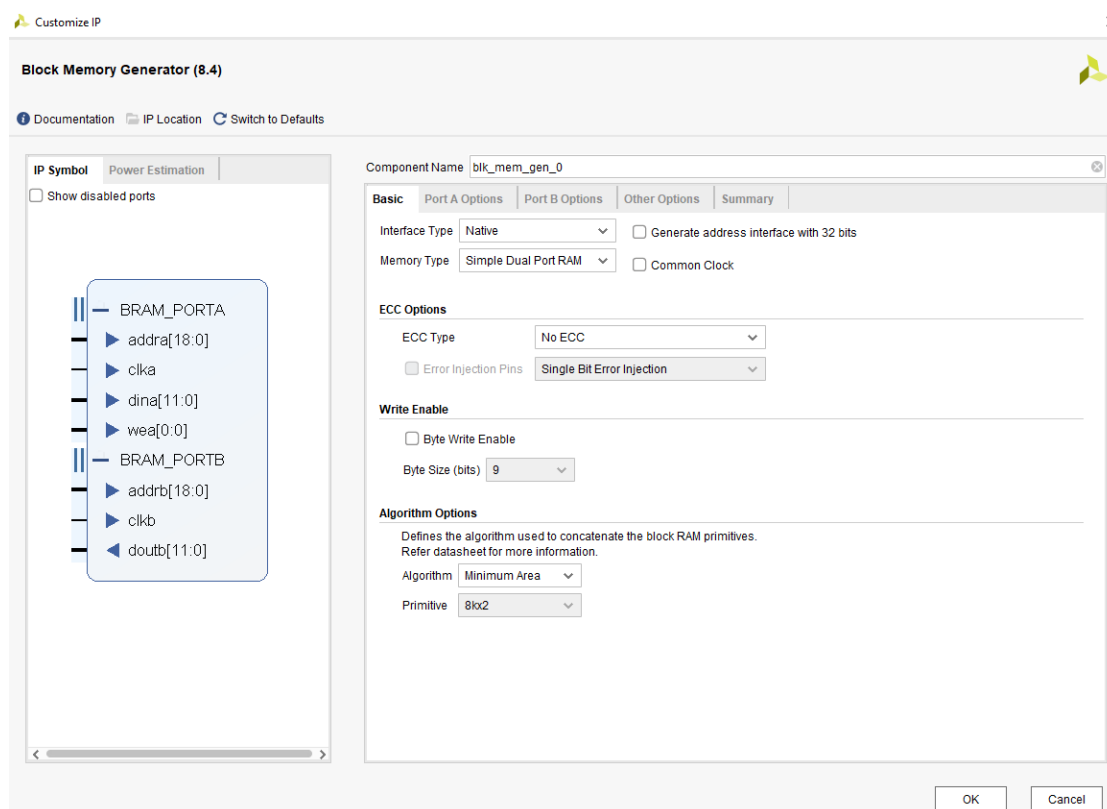


Figura 4.5 Parámetros configurables de memoria.

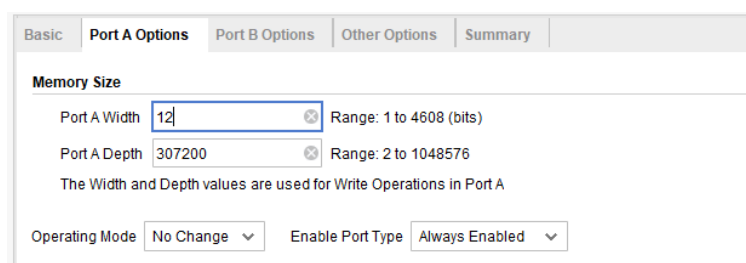


Figura 4.6 Configuración de parámetros para puerto A de la memoria.

Ya configurada la memoria, al módulo *ov7670\_capture.vhd* se le agregaron las señales correspondientes para conectar a la memoria, las cuales son *addr*, *dout* y *we* (*write enable*), la señal *addr* contiene la dirección en la cual se almacenará el dato en *dout*, y *we* se activará cuando se reciba la información completa del pixel (dos ciclos de reloj PCLK), la señal de dirección (*addr*) se reiniciará cuando VSYNC tenga un flanco ascendente, esté en alto.

El módulo *ov7670\_capture.vhd* se muestra en la Figura 4.7 en modo de bloque IP.



Figura 4.7 Módulo encargado de capturar datos del sensor.

Para concluir con este módulo, la entidad en VHDL quedó de la siguiente manera:

```
entity ov7670_capture is
    Port ( pclk   : in    STD_LOGIC;
          vsync  : in    STD_LOGIC;
          href   : in    STD_LOGIC;
          d      : in    STD_LOGIC_VECTOR (7 downto 0);
          addr   : out   STD_LOGIC_VECTOR (18 downto 0);
          dout   : out   STD_LOGIC_VECTOR (11 downto 0);
          we     : out   STD_LOGIC);
end ov7670_capture;
```

### 4.3. Diseño del módulo VGA para visualización de la imagen

Para el diseño del controlador de visualización o despliegue en un monitor VGA, se presenta inicialmente una breve introducción de su funcionamiento y después se detallará el módulo creado para la salida VGA con que cuenta la tarjeta de desarrollo *ZedBoard*.

El estándar de video VGA (*Video Graphics Array*) funciona a través de señales analógicas, por lo que es necesario utilizar Convertidores Digital a Analógico (CDA). El estándar VGA trabaja a una resolución espacial de 640 x 480 pixeles, un formato de color RGB, compartiendo características con el sensor OV7670.

#### 4.3.1. Funcionamiento de la interfaz VGA

Para la comunicación del puerto VGA a un monitor, existen dos señales de sincronía las cuales manejan el espacio horizontal y vertical del monitor, *hsync* y *vsync* respectivamente. La señal *hsync* indica cuando se debe de pasar a la siguiente línea en dirección vertical del monitor.

De igual manera, *vsync* indica cuando inicia y termina por completo una imagen o cuadro, esta información se puede apreciar en la Figura 4.8 [53].

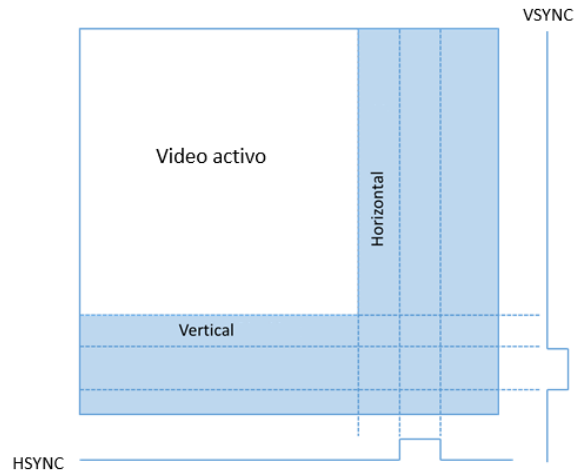


Figura 4.8 Comportamiento de señales de sincronización HSYNC y VSYNC [53].

#### 4.3.2. Modulo VGA en VHDL

Como la comunicación VGA es un protocolo muy usado en aplicaciones con bajo requerimiento en resolución espacial, existe código de acceso libre para el manejo de este protocolo [51], [53], [54], para adaptarlo al hardware específico con el que se trabaja, en este caso el de la tarjeta *ZedBoard*.

El funcionamiento del módulo VGA, *vga.vhd*, se basa básicamente en el comportamiento de las señales de sincronización antes descritas. La entidad *vga.vhd* se muestra a continuación, así como una descripción de cada una de las señales relacionadas:

```
entity vga is
  Port (
    clk25      : in  STD_LOGIC;
    vga_red    : out STD_LOGIC_VECTOR(3 downto 0);
    vga_green  : out STD_LOGIC_VECTOR(3 downto 0);
    vga_blue   : out STD_LOGIC_VECTOR(3 downto 0);
    vga_hsync  : out STD_LOGIC;
    vga_vsync  : out STD_LOGIC;
    frame_addr : out STD_LOGIC_VECTOR(18 downto 0);
    frame_pixel : in  STD_LOGIC_VECTOR(11 downto 0)
  );
```

```
end vga;
```

El módulo trabaja a una frecuencia de 25 MHz mediante la señal *clk25*. Las señales de salida *vga\_green*, *vga\_red*, *vga\_blue* son los valores de los pixeles enviados al monitor, la entrada de los pixeles *frame\_pixel* contiene la información del pixel en formato RGB565 de ahí la dimensión de 12 bits en la señal, el módulo descompone la señal RGB565 en RGB444 debido a que el convertidor digital a análogo con el que cuenta la tarjeta *ZedBoard* es de 4 bit de entrada por color como lo muestra la hoja de datos [42], tal que la descomposición del pixel en sus colores se realiza con:

```
vga_red    <= (frame_pixel(11 downto 8));  
vga_green  <= (frame_pixel( 7 downto 4));  
vga_blue   <= (frame_pixel( 3 downto 0));
```

Las señales *vga\_hsync*, *vga\_vsync* son las encargadas de la sincronía horizontal y vertical, y la señal *frame\_addr* es la señal que enviará la dirección de memoria al bloque RAM (puerto B) para acceso a los pixeles guardados.

#### 4.4. Integración de los módulos

Utilizando los módulos previamente descritos, se utilizó el diseño a bloques en Vivado para la integración de cada uno de los módulos y así consolidar un sistema capaz de capturar datos y mostrarlos en un monitor en tiempo real. Cada una de las señales de los módulos y submódulos fueron creadas con nombres que relacionaran las conexiones entre sí, quedando solo por agregar módulos de generación de señales de reloj (50 MHz y 25MHz) proporcionados por Vivado en forma de IP.

El sistema integrado en la sección FPGA (PL) está plasmado en la Figura 4.9, se pueden observar las salidas y entradas que se conectan a la cámara y al puerto VGA. El mapeo de las salidas y entradas en el FPGA se encuentran en el archivo de restricciones, *.xdc*, en el archivo se definen las entradas, quedando declaradas los nombres de las señales que irán conectadas a los pines del FPGA, la cámara y el VGA. Los pines y su ubicación se encuentran en la hoja de datos proporcionada por el fabricante, de igual manera, el fabricante ofrece un archivo *.xdc* donde se encuentran definidos los pines de cada uno de los componentes integrados en la tarjeta *ZedBoard*, con posibilidad de modificarlo para un acoplamiento a nuestro diseño

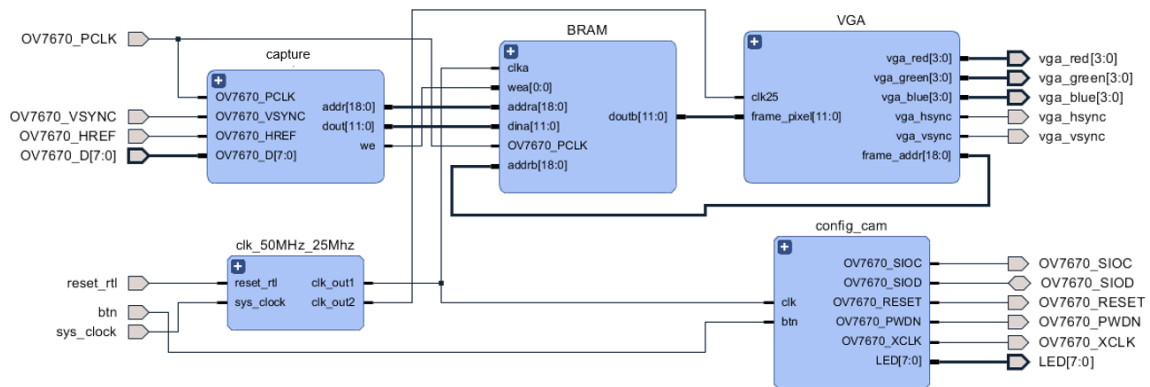


Figura 4.9 Sistema integrado en FPGA.

En las siguientes líneas de código se muestra como quedaron las señales definidas para el periférico VGA:

```

set_property PACKAGE_PIN Y21 [get_ports { vga_blue[0] } ]
set_property PACKAGE_PIN Y20 [get_ports { vga_blue[1] } ]
set_property PACKAGE_PIN AB20 [get_ports { vga_blue[2] } ]
set_property PACKAGE_PIN AB19 [get_ports { vga_blue[3] } ]
set_property PACKAGE_PIN AB22 [get_ports { vga_green[0] } ]
set_property PACKAGE_PIN AA22 [get_ports { vga_green[1] } ]
set_property PACKAGE_PIN AB21 [get_ports { vga_green[2] } ]
set_property PACKAGE_PIN AA21 [get_ports { vga_green[3] } ]
set_property PACKAGE_PIN V20 [get_ports { vga_red[0] } ]
set_property PACKAGE_PIN U20 [get_ports { vga_red[1] } ]
set_property PACKAGE_PIN V19 [get_ports { vga_red[2] } ]
set_property PACKAGE_PIN V18 [get_ports { vga_red[3] } ]
set_property PACKAGE_PIN AA19 [get_ports { vga_hsync } ]
set_property PACKAGE_PIN Y19 [get_ports { vga_vsync } ]

```

Las cuatro componentes del color rojo deben ir asociadas a los pines V20, U20, V19 y V18, del menos al más significativo. Las del verde a los pines AB22, AA22, AB21 y AA21 y las del azul a los pines Y21, Y20, AB20 y AB19. Las señales de sincronía horizontal y vertical van a los pines AA19 e Y19, respectivamente. La alimentación asignada a estos pines se referencia con el código IOSTANDARD tipo LVCMOS33:

```

set_property IOSTANDARD LVCMOS33 [get_ports {vga_blue[*]}]
set_property IOSTANDARD LVCMOS33 [get_ports {vga_green[*]}]
set_property IOSTANDARD LVCMOS33 [get_ports {vga_red[*]}]

```

quedando las demás señales implementadas de igual manera que las anteriores, existiendo solo variaciones en el tipo de IOSTANDARD, para señales tipo LVTTTL y una excepción en la declaración de la señal PCLK la cual se necesita código (*CLOCK\_DEDICATED\_ROUTE FALSE*) para su aceptación como señal de reloj:

```
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets OV7670_PCLK_IBUF]
```

El mapeo de cada uno de los pines en el archivo .xdc queda de la siguiente manera:

```
set_property PACKAGE_PIN Y9 [get_ports {clk100}]; # "GLOBAL CLK"
set_property PACKAGE_PIN Y21 [get_ports { vga_blue[0] } ]
set_property PACKAGE_PIN Y20 [get_ports { vga_blue[1] } ]
set_property PACKAGE_PIN AB20 [get_ports { vga_blue[2] } ]
set_property PACKAGE_PIN AB19 [get_ports { vga_blue[3] } ]
set_property PACKAGE_PIN AB22 [get_ports { vga_green[0] } ]
set_property PACKAGE_PIN AA22 [get_ports { vga_green[1] } ]
set_property PACKAGE_PIN AB21 [get_ports { vga_green[2] } ]
set_property PACKAGE_PIN AA21 [get_ports { vga_green[3] } ]
set_property PACKAGE_PIN V20 [get_ports { vga_red[0] } ]
set_property PACKAGE_PIN U20 [get_ports { vga_red[1] } ]
set_property PACKAGE_PIN V19 [get_ports { vga_red[2] } ]
set_property PACKAGE_PIN V18 [get_ports { vga_red[3] } ]
set_property PACKAGE_PIN AA19 [get_ports { vga_hsync } ]
set_property PACKAGE_PIN Y19 [get_ports { vga_vsync } ]

set_property PACKAGE_PIN Y11 [get_ports { OV7670_PWDN } ]
set_property PACKAGE_PIN AB11 [get_ports { OV7670_RESET } ]
set_property PACKAGE_PIN AA11 [get_ports { OV7670_D[0] } ]
set_property PACKAGE_PIN AB10 [get_ports { OV7670_D[1] } ]
set_property PACKAGE_PIN Y10 [get_ports { OV7670_D[2] } ]
set_property PACKAGE_PIN AB9 [get_ports { OV7670_D[3] } ]
set_property PACKAGE_PIN AA9 [get_ports { OV7670_D[4] } ]
set_property PACKAGE_PIN AA8 [get_ports { OV7670_D[5] } ]

set_property PACKAGE_PIN W12 [get_ports { OV7670_D[6] } ]
set_property PACKAGE_PIN V12 [get_ports { OV7670_D[7] } ]
set_property PACKAGE_PIN W11 [get_ports { OV7670_XCLK } ]
set_property PACKAGE_PIN W10 [get_ports { OV7670_PCLK } ]
set_property PACKAGE_PIN V10 [get_ports { OV7670_HREF } ]
set_property PACKAGE_PIN V9 [get_ports { OV7670_VSYNC } ]
set_property PACKAGE_PIN W8 [get_ports { OV7670_SIOD } ]
set_property PACKAGE_PIN V8 [get_ports { OV7670_SIOC } ]

set_property PACKAGE_PIN T22 [get_ports { LED[0] } ]
set_property PACKAGE_PIN T21 [get_ports { LED[1] } ]
set_property PACKAGE_PIN U22 [get_ports { LED[2] } ]
set_property PACKAGE_PIN U21 [get_ports { LED[3] } ]
set_property PACKAGE_PIN V22 [get_ports { LED[4] } ]
set_property PACKAGE_PIN W22 [get_ports { LED[5] } ]
```

```

set_property PACKAGE_PIN      U19 [get_ports {    LED[6]    }    ]
set_property PACKAGE_PIN      U14 [get_ports {    LED[7]    }    ]

set_property PACKAGE_PIN      T18 [get_ports {    btn    }    ]

# nivel de voltaje
set_property IOSTANDARD LVTTTL [get_ports btn]
set_property IOSTANDARD LVTTTL [get_ports {LED[*]}]
set_property IOSTANDARD LVTTTL [get_ports OV7670_PCLK]
set_property IOSTANDARD LVTTTL [get_ports OV7670_SIOC]
set_property IOSTANDARD LVTTTL [get_ports OV7670_VSYNC]
set_property IOSTANDARD LVTTTL [get_ports OV7670_RESET]
set_property IOSTANDARD LVTTTL [get_ports OV7670_PWDN]
set_property IOSTANDARD LVTTTL [get_ports OV7670_HREF]
set_property IOSTANDARD LVTTTL [get_ports OV7670_XCLK]
set_property IOSTANDARD LVTTTL [get_ports OV7670_SIOD]
set_property IOSTANDARD LVTTTL [get_ports {OV7670_D[*]}]
set_property IOSTANDARD LVCMOS33 [get_ports {vga_blue[*]}]
set_property IOSTANDARD LVCMOS33 [get_ports {vga_green[*]}]
set_property IOSTANDARD LVCMOS33 [get_ports {vga_red[*]}]
set_property IOSTANDARD LVCMOS33 [get_ports vga_hsync]
set_property IOSTANDARD LVCMOS33 [get_ports vga_vsync]
set_property IOSTANDARD LVCMOS33 [get_ports clk100]

set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets OV7670_PCLK_IBUF]

```

#### 4.5. Implementación de algoritmos punto a punto

Al módulo VGA se le agregaron comandos de procesamiento como conversión de imagen RGB a escala de grises, imagen binarizada, negativo de la imagen y detección de bordes mediante el *kernel* de una dimensión (derivada).

Para la conversión a niveles de gris y que la imagen pudiera observarse en el monitor, se realizó la asignación del color verde (color que contiene mayor información de la imagen) a cada uno de los colores de salida, tal que:

```

vga_red    <= frame_pixel( 7 downto 4);
vga_green  <= frame_pixel( 7 downto 4);
vga_blue   <= frame_pixel( 7 downto 4);

```

Para el binarizado de la imagen se aplicó una sentencia IF con un umbral de 8 (“1000”) y se asignó el valor máximo y el mínimo denotado por el resultado de la inferencia IF:

```

if frame_pixel( 7 downto 4) > "0100" then
    vga_red    <= "1111";

```

```

    vga_green <= "1111";
    vga_blue  <= "1111";
else
    vga_red   <= (others => '0');
    vga_green <= (others => '0');
    vga_blue  <= (others => '0');
end if;

```

La imagen binarizada se usó para la detección de bordes en una dimensión captando los cambios de intensidad en las direcciones  $(x,y)$ . Fue necesario guardar información para su análisis de la imagen binarizada; si el pixel anterior es alto, y el pixel nuevo es bajo, se denota la detección de un borde como se muestra a continuación para la dirección horizontal  $x$ :

```

x1 <= x0; --x0 es el valor de la imagen binarizado
if x0 = x1 then
    vga_red   <= (others => '0');
    vga_green <= (others => '0');
    vga_blue  <= (others => '0');
else
    vga_red   <= "1111";
    vga_green <= "1111";
    vga_blue  <= "1111";
end if;

```

De igual manera se aprovechó el cómputo que ofrece el diseño en VHDL con la compuerta NOT para la obtención del negativo del valor del pixel:

```

vga_red   <= NOT (frame_pixel(11 downto 8));
vga_green <= NOT (frame_pixel( 7 downto 4));
vga_blue  <= NOT (frame_pixel( 3 downto 0));

```

Con los elementos de componentes arriba descritos, el esquema final de procesamiento punto a punto de imágenes se muestra en la Figura 4.10, mostrando los diferentes bloques diseñados.

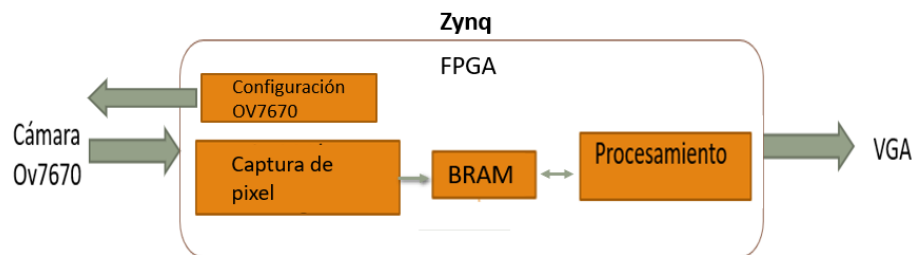


Figura 4.10 Esquema de sistema en FPGA.

El sistema muestra los elementos primarios de adquisición, almacenamiento, procesamiento y despliegue de la imagen. El sistema trabaja con un flujo de datos de entrada a nivel pixel y el procesamiento hasta este momento es también a nivel pixel, excepto en la detección de bordes donde es necesario comparar el valor de dos pixeles para generar el pixel de salida. La salida aun cuando es pixel por pixel, el usuario observa la imagen completa. En realidad, este sistema tiene la capacidad de procesar video (secuencia de imágenes en el tiempo) y el resultado del procesamiento se observará en el video de salida a 30 cuadros por segundos.

## CAPÍTULO V

### VIVADO HIGH LEVEL SYNTHESIS (HLS)

Las herramientas HLS presenta una idea de desarrollo vista antes a través del tiempo, recordando como la programación en sus inicios se realizaba con lenguaje de máquina y se simplificó con lenguaje ensamblador, posteriormente, el lenguaje ensamblador fue remplazado por lenguajes llamados de alto nivel como el C que aplican una mayor abstracción. La programación de C/C++, a menudo representa la forma de pensar del cerebro de forma secuencial sobre un algoritmo y con una representación más digerible que el lenguaje de descripción de hardware (HDL, VHDL o Verilog). Los beneficios de usar un lenguaje de alto nivel quedan representados cuando el sistema a diseñar es grande y complejo. En un programa clásico en C es “sencillo” visualizar la funcionalidad de un fragmento de código, en qué orden es la ejecución de sus sentencias y con un nivel de abstracción en el cual el usuario no se tiene que preocupar por el proceso que se llevará a cabo en el CPU. En los lenguajes HDL el usuario debe tener en cuenta constantemente los procesos que se ejecutan en paralelo, por lo tanto, en sistemas grandes se puede complicar el seguimiento de lo que sucede en instantes de tiempos definidos [55], [56].

Dicho lo anterior, las herramientas HLS ayudan a que los diseños de sistema sean más fáciles de modificar y/o reparar, debido a que es algo trivial el agregar o eliminar código en un programa en C/C++, en comparación con cambiar un detalle en el funcionamiento de un módulo en un diseño HDL.

#### 5.1. Herramientas HLS

El comienzo de las herramientas HLS se encuentra en la década de 1970, en investigaciones principalmente, siendo hasta la década de 1980 y concretándose en los años de 1990 cuando empezó a tener éxito comercial [57], [58]. Sin embargo, las herramientas existentes en épocas anteriores estaban dirigidas a usuarios que estaban familiarizados con la síntesis de diseños en RTL (nivel de transferencia de registro, *Register-Transfer Level*) con lenguajes HDL, quienes realmente no las usaban a menos que presentaran una reducción significativa en el tiempo de

diseño con resultados similares o de mejor calidad. “En general, las herramientas HLS ofrecían resultados que a menudo eran variables e impredecibles, con un mal entendimiento en los factores de control y a los diseñadores se le dificultaba el obtener resultados buenos llegando a pensar en una mala inversión de tiempo” [57], [58].

Las ventas de herramientas HLS registraron un mínimo histórico a principios de la década del año 2000 y se mantuvieron así durante algunos años hasta la próxima generación. Cuando llegó la tercera generación de herramientas de síntesis de alto nivel, las ventas tuvieron un rápido aumento esto alrededor del 2004 y esta vez la gran diferencia fue que la mayoría de las herramientas HLS comenzaron a apuntar hacia lenguajes como C y Matlab. Los usuarios no necesitaban experiencia previa en HDL para utilizar estas herramientas y podían centrarse en lo que ya hacían mejor: algoritmos y diseño del sistema. Además, la calidad de los resultados también mejoró mucho, aumentando aún más el éxito. Hoy en día todavía se ve que el éxito y el desarrollo siguen siendo fuertes, lo que lleva a una multitud de diferentes herramientas de diseño disponibles en el mercado. Algunas de las herramientas HLS en el mercado son Matlab, LabVIEW, Catapult-C, BlueSpec, Synphony, Cynthesizer y la herramienta utilizada en el desarrollo de esta tesis: “Vivado HLS” de Xilinx, anteriormente conocida como AutoPilot antes que de que Xilinx adquiriera a la compañía AutoESL Design Technologies a inicios del 2011.

## **5.2. Diseño con “Vivado HLS”**

La filosofía de diseño proporcionada por Xilinx inicialmente indica que con más restricciones o limitaciones se implementen en código C/C++, será más difícil optimizar el compilador HLS. Por lo tanto, el diseño se puede realizar con ventaja dentro de una función, ya que el código se puede acelerar considerando como única función llamada desde un programa.

Vivado ofrece la herramienta HLS para convertir un lenguaje de alto nivel como C/C++ a un lenguaje descriptivo VHDL o Verilog envuelto en un bloque IP (*Intellectual Property*). El diseño de hardware con esta herramienta es rápido y puede ser usada sin tener conocimiento en un lenguaje descriptivo. Los bloques diseñados pueden ser instanciados con lógica programable o en un diseño de bloques en Vivado, la comunicación con los bloques IP diseñados es por medio de interfaces AXI ya sea desde la parte PL o PS del sistema diseñado. El IP generado no

se puede llamar desde un programa en C/C++ de la manera común en la que se conoce, sin embargo, se pueden y deben usarse pruebas para corroborar el funcionamiento del IP generado. En Vivado HLS se puede simular el sistema diseñado en archivos C/C++ llamados *Test Bench* para observar la funcionalidad tal y como se hace en la simulación de sistemas desarrollados en VHDL o Verilog dejando la complejidad a un lado. A continuación, se mostrarán los pasos simples que ayudan a realizar un diseño en Vivado HLS:

1.- En Vivado HLS, crear un proyecto con el archivo *.c/.cpp* de la función (como cargar imagen, multiplicación, etc) de un algoritmo real para su aceleración, se anexan los archivos de inclusión y declaración de funciones (*.h*).

2.- Crear el archivo de simulación *Test Bench .c/.cpp* con la función *main* del algoritmo real donde se aplica la función acelerada en el paso 1 y agregarla al proyecto.

3.- Sintetizar en Vivado HLS la función original a acelerar dada en el paso 1, donde el sintetizado es la implementación de la configuración de las celdas lógicas y recursos de interconexión necesarias del sistema desarrollado.

4.- En este paso se probará la función sintetizada en el archivo *Test Bench* en el cual se incluyen los archivos *.h* creados a base del algoritmo real donde se incluyen librerías como OpenCV (OpenCV se tratará más adelante).

5.- Observar los resultados en la carpeta de simulación del proyecto que crea Vivado HLS.

Los pasos antes mencionados se pueden resumir en un esquema tal y como se muestra en la Figura 5.1, donde se observa como la función es llevada a Vivado HLS para su aceleración, y no todo el programa en lenguaje C/C++.

Se puede asumir que uno de los archivos más importantes es el archivo de *Test Bench* ya que nos muestra el comportamiento de nuestra función a emplear en el diseño PL-PS del sistema. Xilinx muestra que el no usar un archivo *Test Bench* antes de implementar la función es un error que cometen los usuarios en el flujo del diseño de HLS [59].

Si se tiene éxito con la etapa de sintetización, se genera un informe de latencia y si se selecciona, un informe de forma de onda para realizar un estudio más detallado

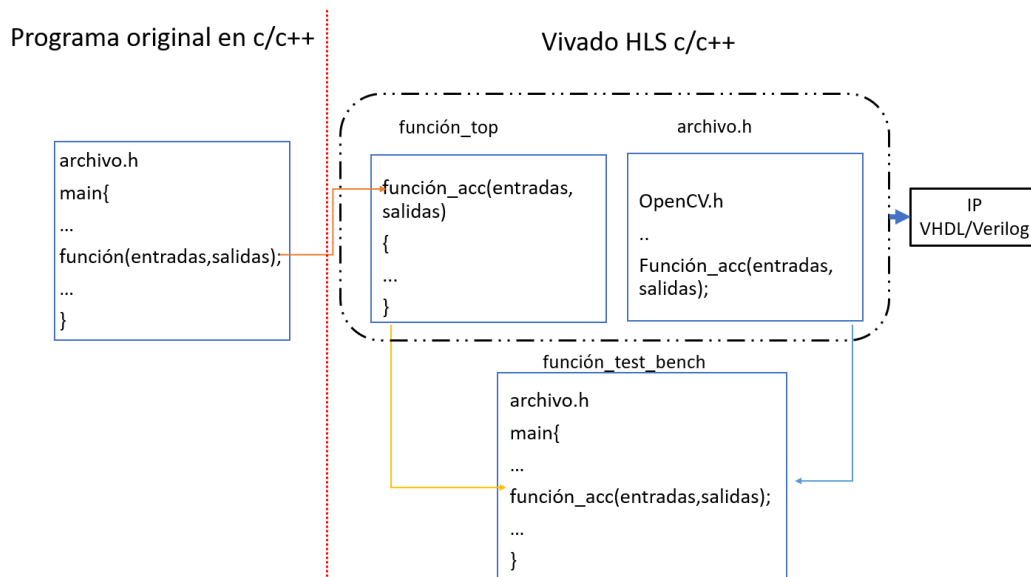


Figura 5.1 Diseño en Vivado HLS para la aceleración de funciones.

### 5.2.1. Optimización de código y sintetización

Cuando el informe de la función sintetizada no es el deseado, existen procedimientos para mejorar y llegar a una optimización del diseño sintetizado, creando la IP que satisfaga la demanda del usuario. La ganancia de rendimiento que se obtiene de la transformación de un programa C/C++ a HDL que se ejecuta en un FPGA proviene de la herramienta HLS, en este caso Vivado HLS, quien se enfoca a analizar las características del código y llegar a elementos tales como:

- Acotamiento (*Flatten*) de ciclos, estructuras de datos como matrices y arreglos para una mejor ejecución paralela y menos sobrecarga.
- Desenrollar (*Unroll*) ciclos y ejecutar varias o todas las iteraciones simultáneamente.
- Unión (*Merge*) de ciclos para obtener una latencia y sobrecarga menor.
- Canalizar (*Pipeline*) los ciclos y funciones usadas para aumentar la eficiencia.

Las directivas determinan cuanto afectará el código a dicha función y en que partes del código afectar. También definen el tipo de interfaz que tendrá la función acelerada PL-PS, de lo contrario se pueden producir cuellos de botella si se elige una interfaz con poco ancho de banda. Este proceso de preparación se realiza principalmente al "etiquetar" partes de código con las directivas, que instruyen al compilador sobre cómo sintetizar esa parte de código en HDL. La directiva generalmente se especifica utilizando *pragmas* [60]. En la programación convencional de C/C++, los *pragmas* son oraciones especiales que controlan el comportamiento del compilador, es decir son directivas de compilador. Un uso común es *#pragma* una vez colocado en la parte superior de un archivo de encabezado para indicar al compilador que solo incluya ese archivo de encabezado una vez [61]. Vivado HLS tiene como objetivo agilizar el proceso de colocación de directivas al proporcionar a los usuarios opciones de GUI para insertar dichos pragmas en ubicaciones clave en el código [62]. El efecto de las diferentes directivas utilizadas para optimizar los bucles es el siguiente.

- *Loop Unrolling* (Desenrollar un ciclo o bucle). Al desenrollar un ciclo se ejecutarán varias iteraciones simultáneamente. Se puede desenrollar parcialmente, lo que significa que todo el ciclo se completa en grupos concurrentes de  $N$ , donde  $N$  es un número especificado por el usuario.  $N = 3$  indica tres iteraciones del bucle completadas simultáneamente, y luego las tres siguientes se realizan simultáneamente, etc. Un ciclo completamente desenrollado es cuando  $N = \text{max}$  iteraciones del ciclo, lo que hace que todas las iteraciones del lazo se ejecuten en un solo ciclo de reloj. La configuración predeterminada en Vivado HLS es dejar los ciclos sin cambios (enrollados). Esto se debe a que desenrollar un bucle está directamente relacionado con la cantidad de recursos del FPGA que se utiliza. Ejecutar un bucle desenrollado donde  $N = 2$  significa lograr el doble de la velocidad, pero también el doble del costo de los recursos. Como se puede imaginar, un bucle completamente desenrollado puede reclamar una gran cantidad de recursos si es grande Figura 5.2. Si una iteración de un bucle depende del resultado de una iteración anterior, no es posible desenrollar el bucle [59]. Esta directiva recuerda cómo

funcionan los bucles en VHDL, donde todas las iteraciones de un bucle se ejecutan en un solo ciclo de reloj [63].

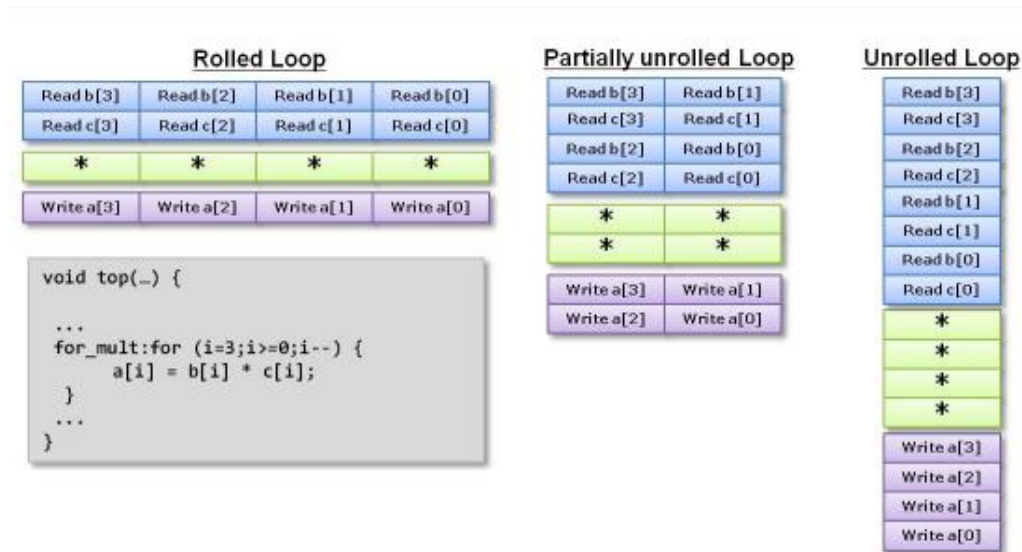


Figura 5.2 Ejemplos de diferentes síntesis del ciclo definido en cuadro gris, “Rolled Loop” es realizado en 4 ciclos de reloj, y se usan pocos recursos, mientras que los ejemplos desenrollados usan más recursos para la ejecución de 2 y 1 ciclo de reloj, respectivamente [59].

- *Loop flattening* (Acotamiento de ciclo). El FPGA tarda un ciclo de reloj para entrar en un ciclo y un ciclo de reloj para salir de un ciclo. Puede que no sea una gran diferencia de latencia cuando se fusionan dos ciclos separados, pero cuando los ciclos están anidados, tiene una importancia mucho mayor. Tomando como ejemplo un ciclo externo que se ejecuta durante 100 iteraciones, con un ciclo interno que ejecuta 5 iteraciones, tomaría un total de 200 ciclos de reloj gastados en entrar y salir del bucle interno. El acotamiento del ciclo ayuda a resolver este problema al eliminar el ciclo interno y mover el cuerpo del ciclo al ciclo externo, lo que reduce significativamente la sobrecarga de ejecución del ciclo [59], [64].
- *Loop pipelining* (Canalizado de ciclo). En ocasiones, los ciclos contienen dependencias de datos que impiden que el ciclo se desenrolle completamente. Otra opción podría ser utilizar la canalización o *pipelining*. La Figura 5.3 ilustra un ejemplo de *pipelining* en un ciclo. Sin *pipelining*, se requieren tres ciclos de reloj para las operaciones de lectura (RD) y seis ciclos de reloj en total para que termine todo el ciclo. Sin embargo, al utilizar *pipelining* en el ciclo, otro ciclo de lectura puede

comenzar una vez que finalice el primero, y todo el proceso puede terminar en solo cuatro ciclos. Una medida importante para la canalización de bucles es el intervalo de iniciación [63]. Para hacer *pipelining* un ciclo, se coloca al comienzo del ciclo la directiva `#Pipelining HLS`.

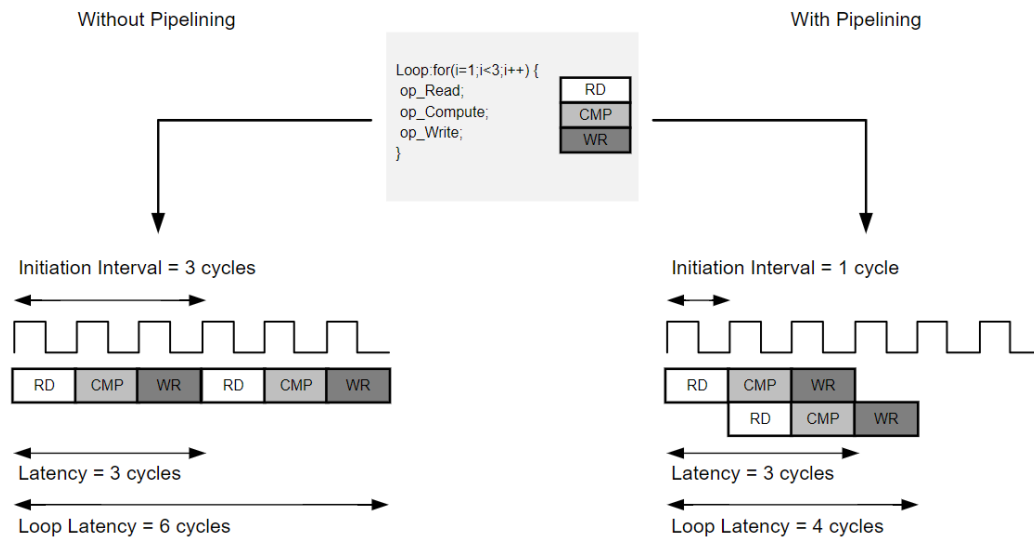


Figura 5.3 Pipelining en un ciclo [63].

- *Loop Dataflow Pipelining* (Canalización de flujo de datos en bucle). Opera en un alcance mayor. Realiza *Pipelining* a múltiples ciclos separados para ejecutar sus iteraciones independientemente de los otros ciclos. Esta directiva también funciona de la misma manera para una serie de funciones. La mayor restricción es que las variables deben ser producidas por un ciclo/función y consumidas por otro ciclo/función, de acuerdo a la Figura 5.4.

Como se mencionó, el paralelismo entre las iteraciones del ciclo está limitado por las dependencias de datos entre las iteraciones del ciclo y los recursos de hardware disponibles. Una dependencia de datos de una operación durante una iteración, a otra operación durante una iteración consecutiva a la primera, se denomina dependencia transportada en el ciclo. Esto implica que la segunda operación en la iteración consecutiva no puede ejecutarse antes de la primera operación en la primera iteración, ya que la primera operación proporcionará la entrada a la siguiente operación de iteración. Las dependencias transportadas en bucle son el factor

limitante para lograr intervalos de iniciación cortos. Los accesos de matriz son una fuente común de dependencias transportadas en bucle.

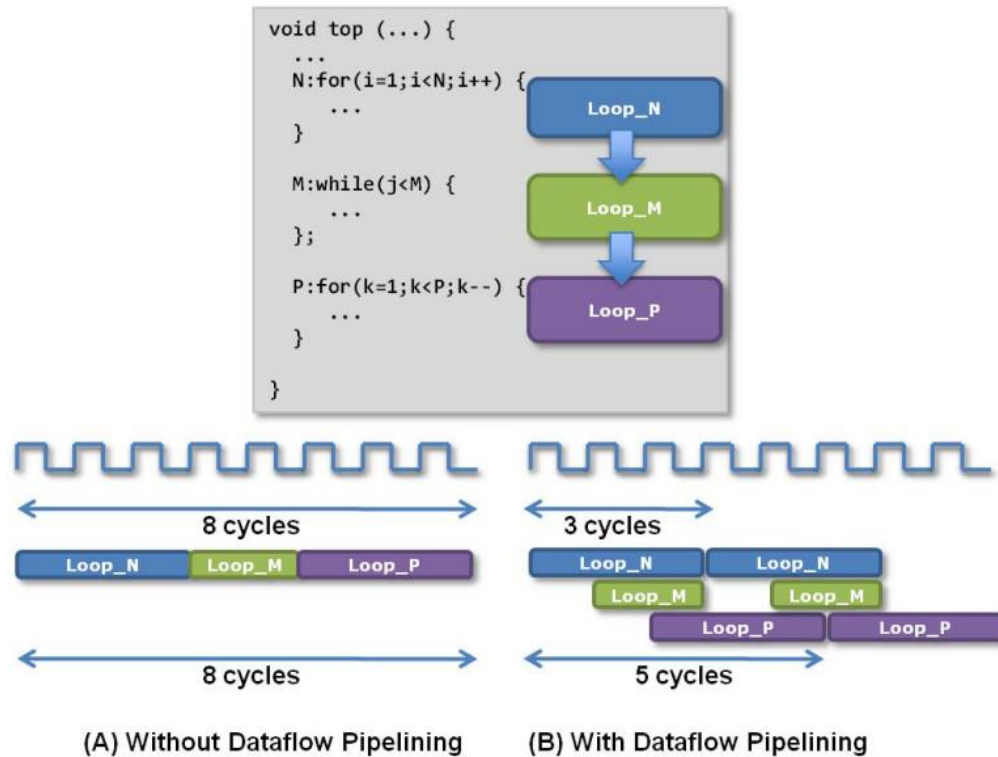


Figura 5.4 Los ciclos separados se les aplica pipelining para que cada ciclo consecutivo pueda ejecutar sus iteraciones independientemente [59].

### 5.3. Librería OpenCV para Vivado HLS

Muchos sistemas de visión por computadora se implementan o al menos se prototipan utilizando la biblioteca de procesamiento de imágenes de código abierto OpenCV [65]. OpenCV contiene implementaciones comúnmente utilizadas y optimizadas de muchas técnicas de visión por computadora [66]. Sin embargo, OpenCV solo se dirige a procesadores de escritorio y GPU. A pesar de que muchas aplicaciones en la biblioteca OpenCV han sido muy optimizadas para permitir que las aplicaciones se ejecuten cerca del tiempo real, una versión de implementación integrada optimizada aún sería preferible. Debido a esto, Xilinx incluye una biblioteca OpenCV en Vivado HLS *Test Bench* y en Vivado SDK. Además, existe una versión simplificada sintetizada de OpenCV que se utiliza en este trabajo, llamada *HLS Video Library* para su uso en

soluciones de *hardware* en Vivado HLS [65], [67]. La biblioteca está optimizada para rendimiento máximo cuando se implementa en un IP de procesamiento de imágenes HLS. Todo esto significa que es posible utilizar la biblioteca completa de OpenCV en *Test Bench* para realizar una carga y guardado de imágenes. Estas imágenes se pueden enviar y recibir desde la función principal de la solución de hardware, para probar la aplicación implementada de hardware siguiendo el orden de diseño previamente mencionado

### 5.3.1. Librería HLS de video (*HLS Video Library*)

La librería HLS de video viene con limitaciones, que pueden no ser obvias al principio. La biblioteca consta de solo 44 funciones de procesamiento de video, donde la mayoría de ellas tiene una contraparte OpenCV y se pueden encontrar en [68]. Para aclarar, estas son las únicas funciones de procesamiento de imágenes que se pueden sintetizar e implementar en el FPGA. Si no existe una función deseada el usuario la tendrá que implementar. Existen varias diferencias de implementación en la biblioteca de videos HLS en comparación con la biblioteca OpenCV. Muchas funciones de OpenCV utilizan asignación de memoria dinámica, como la declaración de una matriz de tamaño arbitrario, que es un formato para almacenar imágenes. La asignación de memoria dinámica no es sintetizable [68], y contribuye a un comportamiento no determinista que es un rasgo indeseable en los sistemas en el área de seguridad. Otro aspecto de la librería HLS de videos es que modela las imágenes como secuencias de píxeles, utilizando el tipo de datos *hls::stream <>*, en lugar de una matriz de píxeles almacenados en la memoria externa. Esto prohíbe el acceso aleatorio y el tipo *cv::Mat <>* que es el más usado en diseño con OpenCV, no tiene una función equivalente sintetizable. El acceso a la transmisión también implica que un área de la imagen no puede modificarse sin procesar primero los píxeles vecinos no modificados y que una imagen debe duplicarse para que sea procesada por más de una función. Este aspecto se explica en el Capítulo 6.

Además, dado que las aplicaciones intentan evitar las operaciones de punto flotante debido a su alto costo computacional, la biblioteca de videos HLS admite el reemplazo de tipos flotantes y dobles con estructuras de punto fijo como *ap\_fixed <>* y *ap\_ufixed <>* y dependiendo de la aplicación se puede realizar una optimización de punto fijo. Debido a esto, se puede esperar que

la salida de imagen de la biblioteca sintetizable generalmente no sea precisa en bits a la salida de imagen de su contraparte OpenCV [68].

Dentro de la sección PL, las imágenes se envían mediante el protocolo AXI4-Stream [65]. Como se mencionó antes, es el protocolo más usado para la transmisión de una gran cantidad de datos. Por lo tanto, la biblioteca de video HLS se complementa con las funciones de entrada y salida AXI4-Stream (para la interfaz a los puertos del módulo IP en el PL) y las funciones de la interfaz OpenCV. Debido a la propia implementación personalizada de Xilinx, la biblioteca también contiene estructuras de datos para *hls::Mat*, *hls::Scalar*, *hls::Window* y *hls::LineBuffer*.

El sistema Vivado “High Level Synthesis (HLS)” es sin duda una herramienta muy poderosa para acelerar el diseño de aplicaciones que involucran dispositivos FPGA o SoC-FPGA. Sin embargo, el conocer y aplicar correctamente la herramienta conlleva el estudiarla y sobre todo el conocer los elementos necesarios del desarrollo a realizar para definirlos antes de llegar a la síntesis. La librería OpenCV es la colección de algoritmos y estructuras más utilizada en el desarrollo de aplicaciones de visión por computadora, procesamiento de imágenes y áreas relacionadas. El contar con un pequeño subconjunto de componentes en HLS es también una forma de auxiliar en el desarrollo acelerado de aplicaciones de visión en el chip *Zynq*. Sin embargo, el aplicar HLS y OpenCV en el medioambiente de Vivado implica el conocer e ir conociendo una gran cantidad de detalles de estas herramientas y aplicarlas al contexto del desarrollo de nuestra aplicación.

## CAPÍTULO VI

### DISEÑO DE MÓDULOS EN VIVADO HLS PARA PROCESAMIENTO DE IMÁGENES

En este capítulo se aborda el diseño de los módulos para procesamiento de imágenes con Vivado HLS versión 2019, retomando aspectos vistos en el capítulo 5, donde se muestra la forma correcta de diseño de los algoritmos, así como los protocolos que se manejan para la recepción y envío de datos. También se habla de las modificaciones necesarias en los módulos de la cámara y VGA para el acoplamiento *AXI4-Stream* con los módulos IP diseñados en Vivado HLS.

La generación de los algoritmos siguió el esquema de diseño descrito en el capítulo 5 y de manera simplificada se muestra en el diagrama de la Figura 6.1.

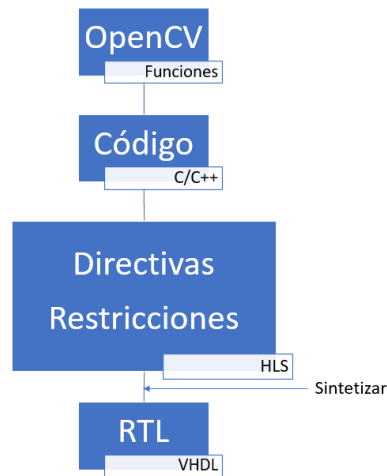


Figura 6.1 Esquema de diseño para módulos IP en Vivado HLS

Los algoritmos obtuvieron resultados óptimos al crear funciones por separado y unirlos en la parte PL, aprovechando la ejecución en paralelo del FPGA. Para terminar, se mostrarán secciones de código y las bases de cada uno de los IP desarrollados, comenzando por Sobel y terminando con la transformada de Hough.

### 6.1. Algoritmo de Sobel para Detección de Bordes Modificado a Ocho Direcciones

Como se comentó en el capítulo 3, la detección de esquinas es muy importante en el procesamiento de imágenes, en esta tesis se tomó en cuenta el trabajo de Xiangxi en [1] donde se desarrollan las ocho ventanas a convolucionar con la imagen de entrada y la implementación de estas mismas en FPGA.

El desarrollo de la implementación del algoritmo fue mostrar el resultado de dos convoluciones desarrolladas en módulos IP y su salida sumada continuamente en una IP de suma. Las convoluciones fueron definidas de 2 en 2, generando 4 IP de convolución (*image\_filter*, *image\_filter\_two*, *image\_filter\_three* e *image\_filter\_four*) y una IP de suma (*image\_filter\_sum*) la cual se replicó 3 veces como se muestra en la Figura 6.2, se utiliza la capacidad de ejecución en paralelo del FPGA.

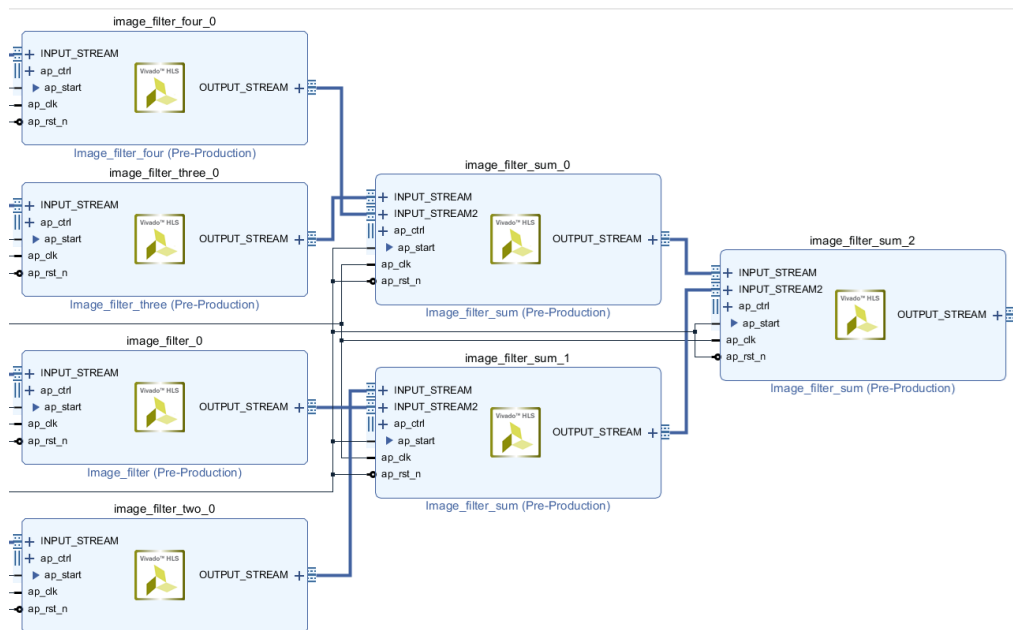


Figura 6.2 Esquema de bloques de la secuencia de datos en el algoritmo de Sobel en 8 direcciones.

A continuación, se describirá el desarrollo de los módulos en Vivado HLS observados en la Figura 6.2. En el capítulo 2 se describió la interfaz AXI4-Stream, por lo tanto, se comenzó con la generación de las interfaces de comunicación para la entrada y salida por medio del protocolo

AXI4-Stream. Para esto, en el archivo *.h* de la función principal del proyecto se definió una secuencia de pixeles del tipo *hls::stream* con una estructura *ap\_ufixed* y se nombró *AXI4-STREAM* como se muestra en el siguiente fragmento de código:

```
typedef hls::stream<ap_axiu<24,1,1,1> >          AXI_STREAM;
```

Los números mostrados en el fragmento de código corresponden a cada una de las señales de la Tabla 2.1 en la sección 2.3.3 del capítulo 2. Para la utilización del flujo de pixeles se utilizó la plantilla *hls::Mat* y se usa para representar una imagen por la librería *hls\_video.h*. La plantilla cuenta con tres parámetros; número de columnas, renglones, numero de canales y tipo de dato. Como se mencionó la dimensión de la imagen es 640x480 y el número de canales son 3 (RGB), la sección de código quedó como se muestra a continuación:

```
typedef hls::Mat<MAX_HEIGHT,    MAX_WIDTH,    HLS_8UC3> RGB_IMAGE; //640 x 480
typedef hls::Mat<MAX_HEIGHT,    MAX_WIDTH,    HLS_12SC1> GRAY_IMAGE; //640 x 480
```

por último, se define la función principal, función que se desarrolla en el archivo *.cpp/.c*:

```
void image_filter(AXI_STREAM& INPUT_STREAM, AXI_STREAM& OUTPUT_STREAM);
```

Ahora se hablará de la función principal *image\_filter* que contiene dos convoluciones asignadas a dos direcciones (0° y 90°) de ocho para la detección de bordes. Las ventanas que se manejan en [1] son de 5x5 pixeles y se definen como constantes tal que:

```
const int kerx[5][5] = {{ 0, -1, 0, 1, 0 }, //
                        { 0, -2, 0, 2, 0 }, //
                        { 0, -4, 0, 4, 0 }, //
                        { 0, -2, 0, 2, 0 },
                        { 0, -1, 0, 1, 0 } };

const int kery[5][5] = {{ 0, 0, 0, 0, 0 }, //
                        { -1, -2, -4, -2, -1 }, //
                        { 0, 0, 0, 0, 0 }, //
                        { 1, 2, 4, 2, 1 },
                        { 0, 0, 0, 0, 0 } };
```

Posteriormente se agregan cada una de las constantes a una plantilla de ventana *hls::Window<5, 5, int>* quedando el código final estructurado como se muestra a continuación:

```
RGB_IMAGE img_0(MAX_HEIGHT, MAX_WIDTH);
GRAY_IMAGE img_1(MAX_HEIGHT, MAX_WIDTH);
...
```

```

GRAY_IMAGE img_9(MAX_HEIGHT, MAX_WIDTH);
RGB_IMAGE img_10(MAX_HEIGHT, MAX_WIDTH);
#pragma HLS dataflow
hls::AXIvideo2Mat(INPUT_STREAM, img_0);
hls::CvtColor<HLS_BGR2GRAY>(img_0, img_1);
//hls::GaussianBlur<3,3>(img_1,img_2);// omitir complejidad
hls::Duplicate(img_1,img_2a,img_2b);
hls::Filter2D(img_2a, img_3, kernelx, ancho);
hls::Filter2D(img_2b, img_4, kernely, ancho);
hls::Zero(img_5);
hls::Zero(img_6);
hls::AbsDiff(img_3, img_5, img_7);
hls::AbsDiff(img_4, img_6, img_8);
hls::AddWeighted(img_7,0.1,img_8,0.1,0.0,img_9);
hls::CvtColor<HLS_GRAY2RGB>(img_9, img_10);

```

donde la función, *CvtColor<HLS\_BGR2GRAY>*, realiza la operación de la ecuación (3.4) descrita en el capítulo 3. Esta función contiene los mismos parámetros y se aplica a cada uno de los píxeles con sus respectivos canales de color RGB. Un aspecto importante es el acomodo de los colores, por lo que hay que tener cuidado cuando se use esta función, incluso la librería permite el uso de, *CvtColor<HLS\_RGB2GRAY>*, donde los colores se posicionan como comúnmente se utilizan.

Si se observa detalladamente el código anterior, se notará que la imagen parece ir a través de un canal de filtros hasta llegar al resultado, pero en el FPGA no es así, debido a que una imagen leída en o desde *INPUT\_STREAM* no se almacena en una variable. Esto significa que en el momento que los píxeles pasen por *hls::CvtColor* inmediatamente pasarán a *hls::Duplicate* y los píxeles posteriores de la imagen se seguirán procesando en *hls::CvtColor*. Este tipo de ejecución es importante para hacer rápido el procesamiento en el FPGA. Esto también significa que los datos, por ejemplo, en *img\_1* solo se puedan usar una sola vez para luego ser consumidos. En este sentido, las variables de imagen se consideran más como identificadoras de una secuencia que como un espacio para almacenar datos. Sin embargo, dado que las funciones *hls::* en realidad están escritas en C/C++, las variables se asignan en la memoria RAM, pero esto es solo para que pueda usarse en simulaciones en C.

En este caso *img\_1*, se utiliza en las funciones de *hls::Filter2D* para las direcciones *x*, *y* siendo de gran utilidad la función *hls::Duplicate* para que se puedan producir dos nuevos identificadores de secuencia. Si se utiliza *img\_1* en dos filtros diferentes habría advertencias que

indican que los datos en *img\_1* ya se habían usado, y si las advertencias se ignoran y se inicia una síntesis, se produciría un error y se detendría el proceso. El resultado de la síntesis arroja el modo en el que los identificadores se ejecutarán como se muestra en la Figura 6.3, haciéndose notar la ejecución dada por *dataflow pipelining*



Figura 6.3 Tiempos de ejecución de la función principal *image\_filter*.

La Figura 6.4 muestra una estimación del rendimiento y recursos utilizados de los bloques desarrollados, utilizando un reloj base de 25 MHz (40 ns). Por lo tanto, se estima que se cumple el tiempo con un margen de 7.64 ns y la implementación realiza un *dataflow pipelining* con una latencia de rendimiento máxima de 314,430 ciclos de reloj, y se puede comenzar a procesar después de un intervalo de 313,797 ciclos. Esto implica un rendimiento de 81 cuadros por segundo aproximadamente ( $1/(314,430 * 40 \text{ ns})$ ).

En la Figura 6.4b también se observa una utilización en recursos de hardware sin limitantes para la implementación. Vivado HLS en el reporte de diseño muestra el uso de 240 BRAM, pero la tarjeta *ZedBoard* cuenta con solo 140, esto expone un error del *software* Vivado HLS.

Una vez con la función creada, se pasó a la sinterización u generación de RTL en VHDL y en forma de IP de acuerdo al flujo en la Figura 6.2.

Timing (ns)

Summary

| Clock  | Target | Estimated | Uncertainty |
|--------|--------|-----------|-------------|
| ap_clk | 40.00  | 32.353    | 5.00        |

Latency (clock cycles)

Summary

| Latency |        | Interval |        | Type     |
|---------|--------|----------|--------|----------|
| min     | max    | min      | max    |          |
| 309139  | 314430 | 309122   | 313797 | dataflow |

Summary

| Name            | BRAM_18K | DSP48E | FF     | LUT   | URAM |
|-----------------|----------|--------|--------|-------|------|
| DSP             | -        | -      | -      | -     | -    |
| Expression      | -        | -      | 0      | 90    | -    |
| FIFO            | 0        | -      | 182    | 770   | -    |
| Instance        | 10       | 31     | 4775   | 14021 | 0    |
| Memory          | -        | -      | -      | -     | -    |
| Multiplexer     | -        | -      | -      | 108   | -    |
| Register        | -        | -      | 18     | -     | -    |
| Total           | 10       | 31     | 4975   | 14989 | 0    |
| Available       | 280      | 220    | 106400 | 53200 | 0    |
| Utilization (%) | 3        | 14     | 4      | 28    | 0    |

Figura 6.4 Estimación en rendimiento y uso de recursos en el módulo *image\_filter*.

Para la corroboración de la función se utilizó el archivo *Test\_Bench* creado en C++ y se hace notar que la escritura del código en esta sección fue utilizando OpenCV de manera normal sin restricciones.

```
#include <hls_opencv.h>
#include "image_filter.h"
#include <iostream>
using namespace std;
int main (int argc, char** argv) {
    int const rows = MAX_HEIGHT;
    int const cols = MAX_WIDTH;
    cv::Mat src = cv::imread("test_in.bmp");
    cv::Mat dst = src;
    AXI_STREAM INPUT_STREAM, OUTPUT_STREAM;
    cvMat2AXIvideo(src, INPUT_STREAM);
    image_filter(INPUT_STREAM, OUTPUT_STREAM);
    AXIvideo2cvMat(OUTPUT_STREAM, dst);

    cv::imwrite("test_out.bmp", dst);

    return 0;
}
```

La metodología fue la misma para la elaboración de los IP restantes, *image\_filter\_two*, *image\_filter\_three* e *image\_filter\_four*, manteniendo los archivos iguales a excepción de la ventana a convolucionar y el nombre de los archivos.

Para dar por concluido este algoritmo, se comentará la función *Top image\_filter\_sum*. La función hace uso de dos entradas de imagen las cuales se sumarán, la función a acelerar se declaró:

```
void image_filter_sum(AXI_STREAM& INPUT_STREAM,AXI_STREAM& INPUT_STREAM2,
AXI_STREAM& OUTPUT_STREAM)
```

y la secuencia de ejecución en los identificadores fue la siguiente:

```
#pragma HLS dataflow
hls::AXIvideo2Mat(INPUT_STREAM, img_0);
hls::AXIvideo2Mat(INPUT_STREAM2, img_1);
hls::CvtColor<HLS_BGR2GRAY>(img_0, img_2);
hls::CvtColor<HLS_BGR2GRAY>(img_1, img_3);
hls::Merge(img_2, img_3, img_4);
hls::CvtColor<HLS_GRAY2RGB>(img_4, img_5);
hls::Mat2AXIvideo(img_5, OUTPUT_STREAM);
```

En la Figura 6.5 se observa cómo es la secuencia de ejecución de los pasos en cada proceso haciéndose notar también el *dataflow pipelining*, donde los procesos pueden ser llamadas de manera paralela para aumentar la velocidad. Los recursos estimados y el desempeño en latencia de ejecución del módulo se hacen notar en la Figura 6.6, indicando valores aceptables con un reloj base de 25 MHz para su implementación en el FPGA (frecuencia mínima permitida por el reloj de captura de la cámara).

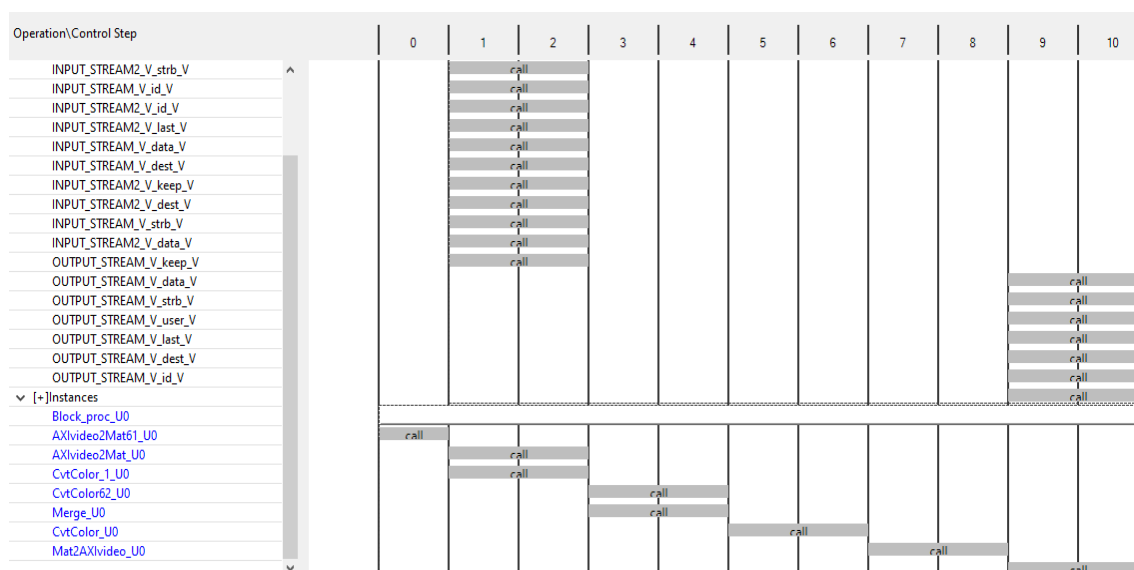


Figura 6.5 Secuencia de procesos y control de pasos generada por Vivado HLS para el módulo de suma de imágenes.

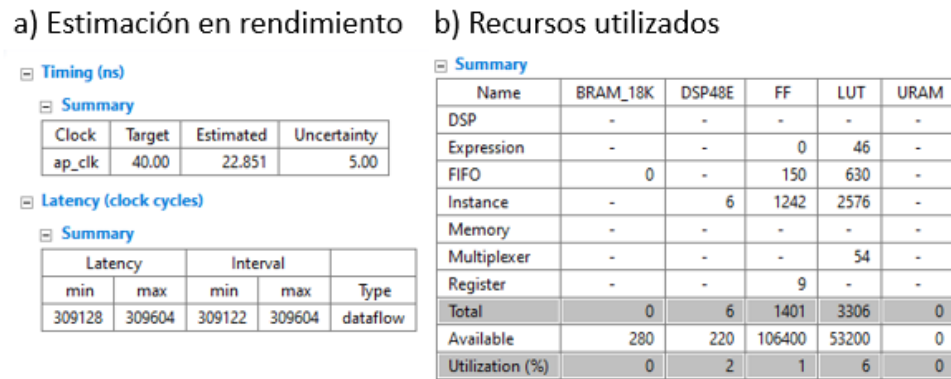


Figura 6.6 Estimación en rendimiento y recursos en módulos image\_sum.

## 6.2. Detección de esquinas por el algoritmo de Harris

Una de las funciones de OpenCV que vienen incluidas es el detector de esquinas Harris. Harris utiliza varios métodos matemáticos descritos en el capítulo 3, y producen una imagen con puntos que marcan la existencia de esquinas. Se decidió utilizar este algoritmo por el nivel de complejidad y así evaluar la herramienta Vivado HLS. La implementación del algoritmo se describe con el flujo de datos (Figura 6.7) correspondiente a la metodología presentada en el Capítulo 3.

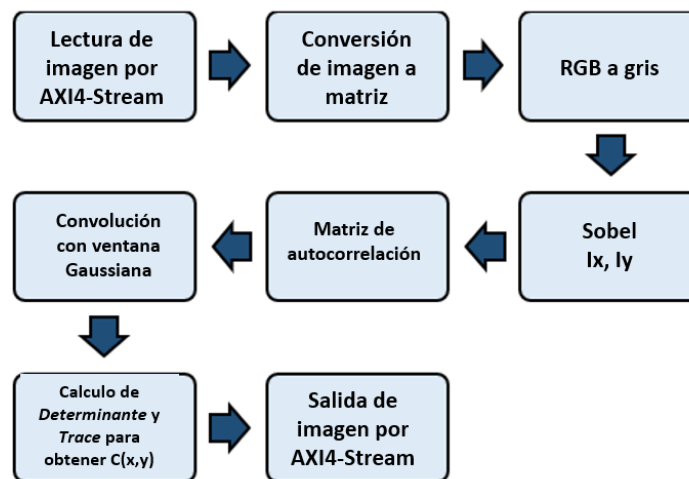


Figura 6.7 Flujo de datos en algoritmo para detección de esquinas.

Por lo tanto, el código mostrado a continuación es en base al flujo del algoritmo planteado:

```

Duplicate(_src,gray1,gray2);

Sobel<1,0,Ksize,BORDERMODEL>(gray1,grad_x);

```

```

Duplicate(grad_x,grad_x1,grad_x2);
Duplicate(grad_x1,grad_x3,grad_x4);
Sobel<0,1,Ksize,BORDERMODEL>(gray2,grad_y);
Duplicate(grad_y,grad_y1,grad_y2);
Duplicate(grad_y1,grad_y3,grad_y4);

Mul(grad_x3,grad_x4,grad_xx);
Mul(grad_y3,grad_y4,grad_yy);
Mul(grad_x2,grad_y2,grad_xy);

BoxFilter<blockSize,blockSize,NORMALIZE,BORDERMODEL>(grad_xx,grad_gx);
BoxFilter<blockSize,blockSize,NORMALIZE,BORDERMODEL>(grad_yy,grad_gy);
BoxFilter<blockSize,blockSize,NORMALIZE,BORDERMODEL>(grad_xy,grad_gxy);

// cim = det - k*(trac)*(trac);
CalcCim(grad_gx,grad_gy,grad_gxy, _dst,k, scale);

```

El algoritmo viene incluido en la función `hls::Harris` de la librería OpenCV para HLS, lo que simplifica su uso en el diseño. La función se describe en su flujo de datos en la Figura 6.7.

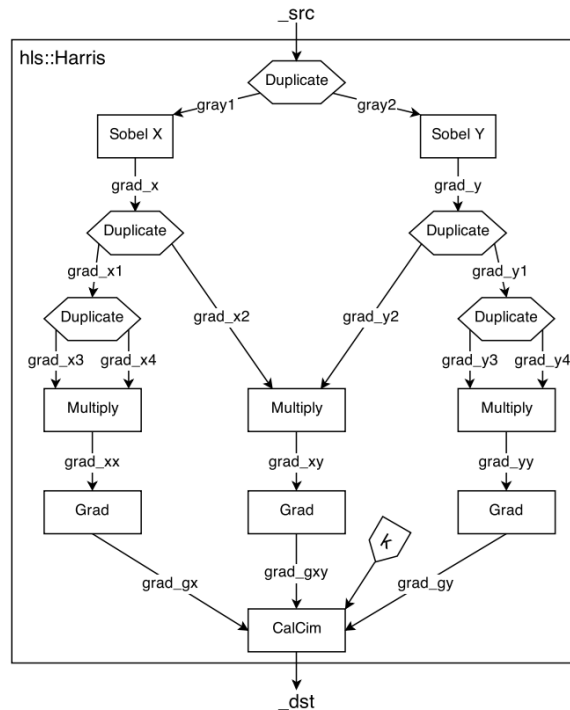


Figura 6.8 Función de Harris en HLS [67].

El código resultante con la implementación como función principal (Top) quedó de la siguiente manera:

```

void harris_accel_hls(AXI_STREAM24& INPUT_STREAM, AXI_STREAM24& OUTPUT_STREAM)
hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC3>      _src(MAX_HEIGHT,MAX_WIDTH);
hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC3>      _dst(MAX_HEIGHT,MAX_WIDTH);
#pragma HLS dataflow
hls::AXIvideo2Mat(INPUT_STREAM, _src);
hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC3>      src0(MAX_HEIGHT,MAX_WIDTH);
hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC3>      src1(MAX_HEIGHT,MAX_WIDTH);
hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC1>      mask(MAX_HEIGHT,MAX_WIDTH);
hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC1>      dmask(MAX_HEIGHT,MAX_WIDTH);
hls::Scalar<3,unsigned char> color(255,0,0);
hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_32FC1>      gray(MAX_HEIGHT,MAX_WIDTH);
hls::CvtColor<HLS_RGB2GRAY>(_src,gray);
hls::Harris<5,3,float>(gray, mask, 0.04, 200);
hls::Dilate(mask,dmask);
//hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC3>      _dmask(MAX_HEIGHT,MAX_WIDTH);
hls::CvtColor<HLS_GRAY2RGB>(dmask,_dst);
//      hls::PaintMask(src1,_dmask,_dst,color);
hls::Mat2AXIvideo(_dst, OUTPUT_STREAM);

```

En este proyecto no se realizaron las definiciones de *hls:Mat* en el archivo *.h* para que se tenga una apreciación más clara en los tipos de datos usados. La imagen de entrada *\_src* es una imagen de 24 bits (3 canales de 8 bits por color), esta imagen se convierte en una imagen de escala de grises flotante de 32 bits, *gray*. El tipo de datos de mayor resolución es necesario durante el cálculo de Harris para evitar que los números se desborden durante operaciones como el cuadrado. Sin embargo, la imagen final de Harris, *mask*, es a escala de grises de 8 bits ya que hay una función de supresión no máxima y umbral incorporada llamada *FindMax*. Esta función marca las características de las esquinas con pixeles blancos (255) cuando se encuentran por encima del umbral designado, y todo lo que está debajo da como resultado un pixel negro (0). Debido a la supresión no máxima, ningún punto blanco será adyacente a otro. Para concluir se utiliza una dilatación de las esquinas encontradas para una mayor apreciación.

Como resultado en la sinterización del algoritmo se tuvo un rendimiento mostrado en la Figura 6.4 con un reloj base de 25 MHz (40 ns). Por lo tanto, se estima que se cumple el tiempo con un margen de 5.039 ns y la implementación realiza un *dataflow pipelining* con una latencia de rendimiento máxima de 319,093 ciclos de reloj, y se puede comenzar a procesar después de un intervalo de 319,088 ciclos. Esto constituye un rendimiento de 78 cuadros por segundo procesados aproximadamente ( $1/(314430 * 40 \text{ ns})$ ).

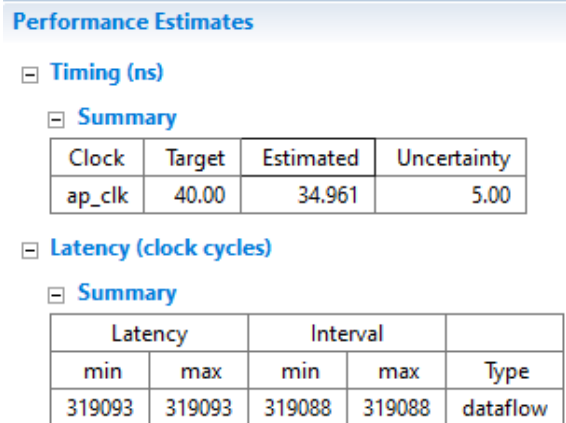


Figura 6.9 Rendimiento estimado en módulo de Harris.

Para dar por culminado el algoritmo de detección de esquinas, se muestra en la Figura 6.10 la secuencia en la cual cada una de las funciones fue llamada y ejecutada para dar una solución apegada al requerimiento del diseño, al igual que el código de simulación *Test Bench* de la función en Vivado HLS:

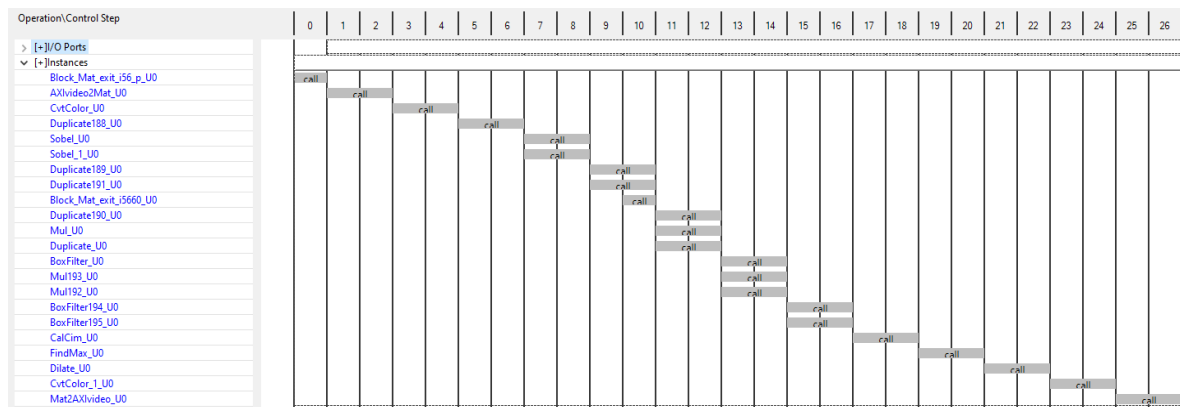


Figura 6.10 Pipeline del algoritmo de Harris para detección de Harris en HLS.

```
#include <hls_opencv.h>
#include "top.h"
#include <iostream>
#include <stdio.h>
#include <opencv2/opencv.hpp>
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"

using namespace std;
```

```

int main()
{
    cv::Mat image_in(MAX_HEIGHT, MAX_WIDTH, CV_8UC3);
    cv::Mat image_out_hls(MAX_HEIGHT, MAX_WIDTH, CV_8UC3);

    image_in = cv::imread("test_1080p.bmp",1);

    AXI_STREAM24 src_axi, dst_axi;
    cvMat2AXIvideo(image_in, src_axi);
    harris_accel_hls(src_axi,dst_axi);
    AXIvideo2cvMat(dst_axi, image_out_hls);
    cv::imwrite("out_harris_points.jpg", image_out_hls);
    return 0;
}

```

### 6.3. Transformada de Hough para Detección de Líneas

A continuación, se muestra el desarrollo del algoritmo en Vivado HLS para la detección de líneas mediante la Transformada de Hough (TH). Al igual que los algoritmos anteriores, se utilizó el archivo *.h* el cual contiene las definiciones de los tipos de imágenes a usar (número de canales y tipos de datos). Cabe mencionar que el algoritmo da la extracción de parámetros  $\rho$ - $\theta$  como resultados de salida debido a los recursos en la tarjeta *ZedBoard*. Por lo tanto, el diseño se basó en el diagrama mostrado en la Figura 6.11.

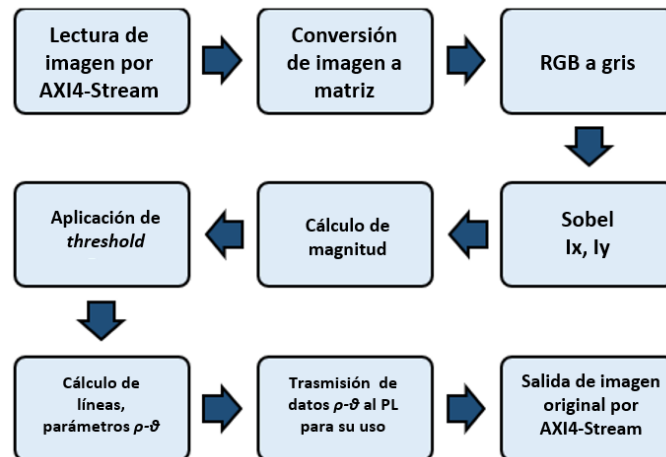


Figura 6.11 Diagrama propuesto para TH en HLS.

El diagrama solo prevé hasta el envío de los datos  $\rho$ - $\theta$  al PL para luego ser enviados por el PS y que este los transmita mediante comunicación serial a Matlab para su análisis y observar su

comportamiento, no obstante, no olvidar que se puede simular este procedimiento mediante el archivo de simulación. A continuación se muestran partes del código implementado en HLS para la detección de líneas con uso de la función *hls::HoughLines2*.

Función Top indicando las entradas y salidas requeridas en el diagrama (Figura 6.10), por lo tanto, el archivo *.h* del diseño HLS incluirá

```
void image_hough(AXI_STREAM_IN& INPUT_STREAM, AXI_STREAM_OUT& OUTPUT_STREAM,
t_lineas_angle lineas_angle[MAX_LINES], t_lineas_rho lineas_rho[MAX_LINES])
```

tal que, se declaran en el archivo *.h* de la siguiente manera las interfaces entrada salida:

```
typedef ap_fixed<16,3,AP_RND> t_lineas_angle;
typedef ap_fixed<16,16,AP_RND> t_lineas_rho;
#define MAX_LINES 30
typedef hls::stream<ap_axiu<24,1,1,1> > AXI_STREAM_IN;
typedef hls::stream<ap_axiu<24,1,1,1> > AXI_STREAM_OUT;
```

El código indica que se utilizará para  $\rho$ - $\theta$  un tipo de dato flotante redondeado infinito positivo (AP\_RDN), un máximo de 30 líneas detectadas por la función *hls::HoughLines2* y un manejo de datos de 24 bits RGB888.

Ahora se describe el flujo de datos en el archivo *.c/.cpp* de la función Top siguiendo el diagrama propuesto en la Figura 6.10:

```
//Crea interfaces AXI streaming
#pragma HLS INTERFACE axis port=INPUT_STREAM
#pragma HLS INTERFACE axis port=OUTPUT_STREAM
#pragma HLS INTERFACE ap_memory port=lineas_angle
#pragma HLS INTERFACE ap_memory port=lineas_rho

    RGB_IMAGE input_img(rows, cols);
    //.
//tipos de imágenes a usar
//
    RGB_IMAGE output_img(rows, cols);
    int num_pixel_por_linea = 500;
    hls::Polar_< t_lineas_angle, t_lineas_rho> lineas_internal[MAX_LINES];

#pragma HLS dataflow
    // 24 Bit (RGB)
    hls::AXIvideo2Mat(INPUT_STREAM, input_img);
    hls::CvtColor<HLS_RGB2GRAY>(input_img, gray_img);
    hls::Duplicate(gray_img, gray_img1, gray_img2);
    hls::Sobel<1,0,3>(gray_img1, sobel_out_hori);
    hls::Sobel<0,1,3>(gray_img2, sobel_out_vert);
```

```

hls::AddWeighted(sobel_out_hori, 1, sobel_out_vert, 1, 0, sobel_out);
hls::Threshold(sobel_out, threshold_image, 20, 255, HLS_THRESH_BINARY);
// hls::Erode(threshold_image, eroded_image); // opcional
hls::Duplicate(threshold_image, hough_in, output_img_gray);
hls::CvtColor<HLS_GRAY2RGB>(output_img_gray, output_img);
hls::HoughLines2<3,5>(hough_in, lineas_internal, num_pixel_por_linea);
// Datos al PS/PL
for (int i = 0; i < MAX_LINES; i++)
{
    lineas_angle[i] = lineas_internal[i].angle;
    lineas_rho[i] = lineas_internal[i].rho;
}
hls::Mat2AXIvideo(output_img, OUTPUT_STREAM);

```

La función usa *hls::HoughLines2* de acuerdo a la manera explicada en el capítulo 3. La plantilla de la función acepta y ofrece los siguientes datos mostrados por su plantilla:

```

template<typename AT, typename RT>
struct hls::Polar_ {
    AT angle;
    RT rho;
};
template<int ROWS, int COLS, int SRC_T, unsigned int theta,
unsigned int rho, typename AT, typename RT, unsigned int linesMax>
void hls::HoughLines2( hls::Mat<ROWS, COLS, SRC_T>&& src,
                      hls::Polar_<AT,RT> (&&lineas)[linesMax],
                      unsigned int threshold );

```

debido a esta plantilla se declararon las señales *t\_lines\_angle* y *t\_lines\_rho* para el manejo de datos en coordenadas polares las cuales son enviadas a una dirección de memoria del PS mediante comunicación AXI de control. El *threshold* es el número de píxeles captados para la que la línea sea regresada, en este caso se manejó un número de 10 para no estresar el FPGA en recursos.

Como resultado en la sinterización del algoritmo se tuvo el rendimiento mostrado en la Figura 6.12 diseñado con un reloj base de 40 ns. Por lo tanto, se estima que se cumple el tiempo de procesamiento con un margen de 5.399 ns y la implementación realiza un *dataflow pipelining* con una latencia de rendimiento máxima de 365,219 ciclos de reloj, y se puede comenzar a procesar después de un intervalo de 365,699 ciclos. Esto constituye un rendimiento de 69 cuadros por segundo procesados aproximadamente ( $1/(365219 * 40 \text{ ns})$ ).

Performance Estimates

Timing (ns)

Summary

| Clock  | Target | Estimated | Uncertainty |
|--------|--------|-----------|-------------|
| ap_clk | 40.00  | 34.601    | 5.00        |

Latency (clock cycles)

Summary

| Latency |        | Interval |        | Type     |
|---------|--------|----------|--------|----------|
| min     | max    | min      | max    |          |
| 365219  | 365699 | 365151   | 365631 | dataflow |

Figura 6.12 Rendimiento estimado para algoritmo de HT.

El flujo de datos generado por el *pragma Dataflow* se muestra en la secuencia ilustrada por la Figura 6.13. El estrés en el algoritmo se obtiene en la función de la TH *hls::HoughLines2*, donde se marcan los picos encontrados por TH en acumuladores que se interconectan al puerto de memoria, haciendo que la función de OpenCV sea costosa en recursos y datos almacenados momentáneamente.

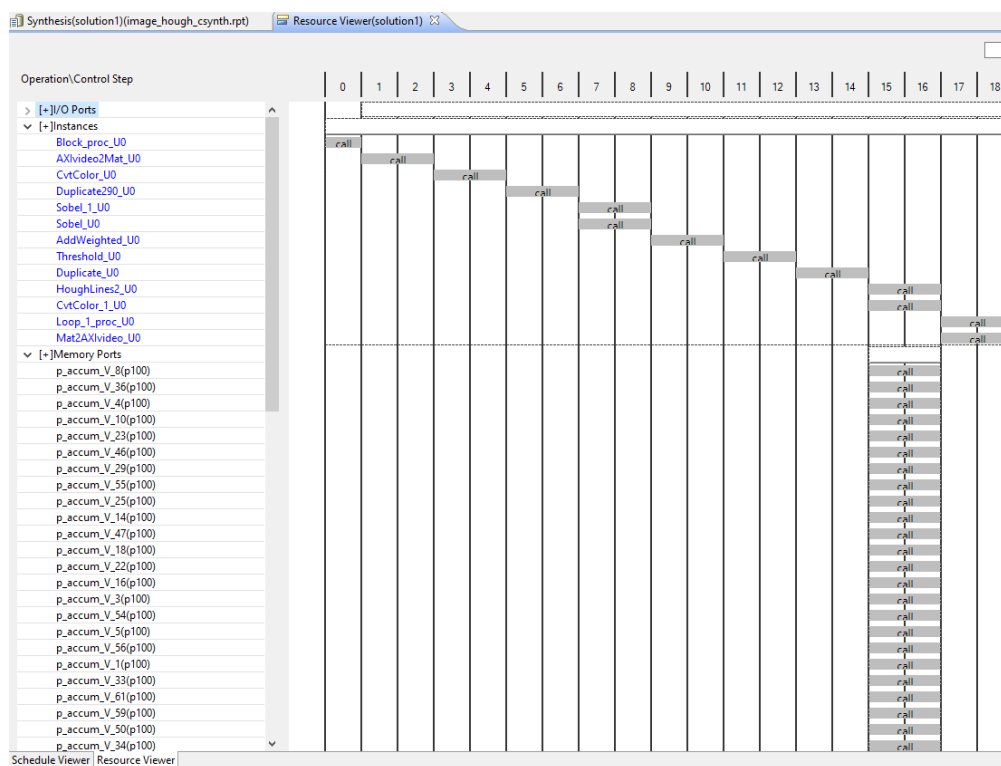
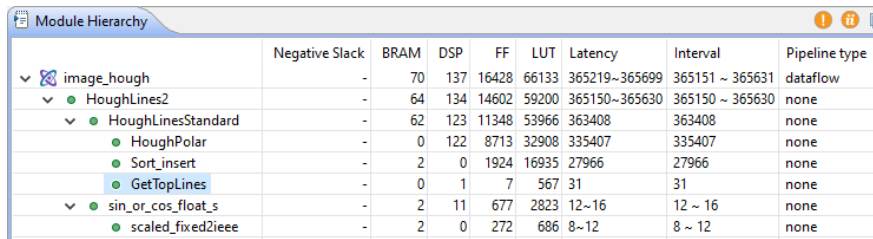


Figura 6.13 Flujo de datos marcados por un Pipelining tipo Dataflow en HLS para la TH.

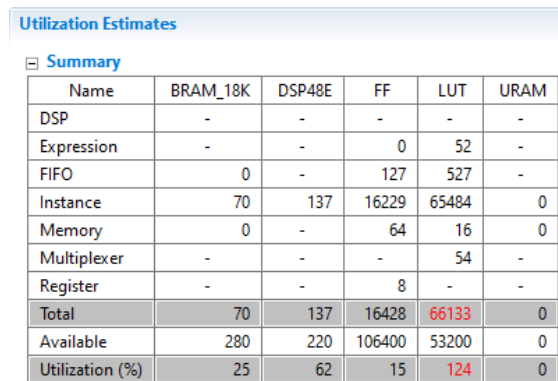
La jerarquía de los procesos en la función *hls::HoughLines2* se muestran en la Figura 6.14 observando que hace uso de las funciones de seno y coseno (así como transformación a punto flotante) tal y como se observó en la explicación de este algoritmo en el capítulo 3.

Por último, se obtienen los picos máximos de los acumuladores para representar una línea los cuales se buscan mediante la función *GetTopLines*. Como se observa en la Figura 6.14, la mayoría del tiempo de procesado es usado por la parametrización de los datos a polar y el ordenamiento de los datos al igual que en memoria, atribuyendo el alto consumo en recursos del FPGA como se observa la Figura 6.15 muestra un desborde del 24% en LUT, recordando que es una estimación dada por Vivado HLS.



|                    | Negative Slack | BRAM | DSP | FF    | LUT   | Latency       | Interval        | Pipeline type |
|--------------------|----------------|------|-----|-------|-------|---------------|-----------------|---------------|
| image_hough        | -              | 70   | 137 | 16428 | 66133 | 365219~365699 | 365151 ~ 365631 | dataflow      |
| HoughLines2        | -              | 64   | 134 | 14602 | 59200 | 365150~365630 | 365150 ~ 365630 | none          |
| HoughLinesStandard | -              | 62   | 123 | 11348 | 53966 | 363408        | 363408          | none          |
| HoughPolar         | -              | 0    | 122 | 8713  | 32908 | 335407        | 335407          | none          |
| Sort_insert        | -              | 2    | 0   | 1924  | 16935 | 27966         | 27966           | none          |
| GetTopLines        | -              | 0    | 1   | 7     | 567   | 31            | 31              | none          |
| sin_or_cos_float_s | -              | 2    | 11  | 677   | 2823  | 12~16         | 12 ~ 16         | none          |
| scaled_fixed2ieee  | -              | 2    | 0   | 272   | 686   | 8~12          | 8 ~ 12          | none          |

Figura 6.14 Jerarquía de funciones en *hls::HoughLines2*.



| Utilization Estimates  |           |            |              |              |          |
|------------------------|-----------|------------|--------------|--------------|----------|
| Summary                |           |            |              |              |          |
| Name                   | BRAM_18K  | DSP48E     | FF           | LUT          | URAM     |
| DSP                    | -         | -          | -            | -            | -        |
| Expression             | -         | -          | 0            | 52           | -        |
| FIFO                   | 0         | -          | 127          | 527          | -        |
| Instance               | 70        | 137        | 16229        | 65484        | 0        |
| Memory                 | 0         | -          | 64           | 16           | 0        |
| Multiplexer            | -         | -          | -            | 54           | -        |
| Register               | -         | -          | 8            | -            | -        |
| <b>Total</b>           | <b>70</b> | <b>137</b> | <b>16428</b> | <b>66133</b> | <b>0</b> |
| Available              | 280       | 220        | 106400       | 53200        | 0        |
| <b>Utilization (%)</b> | <b>25</b> | <b>62</b>  | <b>15</b>    | <b>124</b>   | <b>0</b> |

Figura 6.15 Recursos estimados por HLS en la implementación de la TH.

En consecuencia, los recursos utilizados se muestran como uno de los problemas resultantes en la elaboración de este algoritmo, lo que impide completar el despliegue de resultados al 100% en la plataforma *ZedBoard*. El cómo se llegó a esta conclusión es basada en el diseño del algoritmo para la transformación de los datos resultantes a coordenadas x-y con uso de la función

*hls::cordic::sin\_cos\_range\_redux\_cordic* encontrada en la librería *hls\_math.h* de Vivado HLS. Una vez utilizada la función *cordic* los datos pasan a ser sobrescritos en la imagen original, realizado en un ciclo *for* para cada una de las líneas encontradas. No obstante, el algoritmo no marca error alguno, en síntesis, pero al finalizar la síntesis los recursos utilizados sobrepasaron hasta por 100% los bloques DSP y LUT. Sin embargo, los bloques de memoria aparecen en blanco y no se mapean las funciones señaladas y si no es sintetizada no es posible una simulación. Por esta razón solo se llegó a esta solución, la cual aportó datos suficientes para su comparación con la función de Matlab.

#### **6.4. VDMA y modificación de módulo OV7670 y VGA**

Los módulos diseñados en este capítulo trabajan bajo el protocolo AXI4-Stream y por consecuencia es necesario modificar los módulos *ov7670\_capture* y *vga* desarrollados en el sistema del capítulo 4, los cuales no cuentan con una interfaz de comunicación que permita la transferencia de datos a los módulos de procesamiento digital creados con Vivado HLS. El agregar la interfaz AXI4-Stream al igual que un sistema de resguardo de datos es de suma importancia para la correcta conclusión de este trabajo. Vivado ofrece el IP VDMA que permite guardar información en la memoria DDR (memoria dinámica). A continuación, se explicará la funcionalidad del módulo VDMA y las modificaciones realizadas a los módulos *ov7670\_capture.vhd* y *vga.vhd*.

##### **6.4.1. VDMA**

El bloque IP VDMA de Xilinx se utilizó para transferir cuadros de video a la memoria DDR. El bloque está hecho específicamente para transferir transmisiones de video hacia y desde una ubicación mapeada en memoria [69]. Puede leer un AXI4-Stream y escribirlo en la memoria (*Stream to memory map*, S2MM), y al mismo tiempo leer una ubicación de memoria específica y enviarlo a otro AXI4-Stream (*Memory map to stream*, MM2S). La Figura 6.16 muestra el diagrama de bloques básico de la ruta del video a la memoria con la ayuda de VDMA.

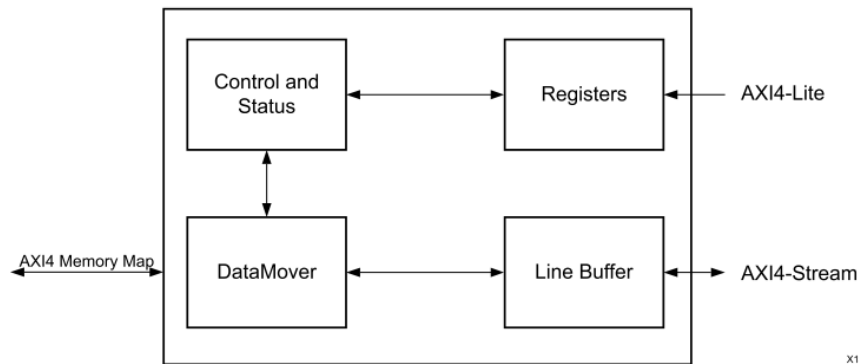


Figura 6.16 Diagrama del IP VDMA [69].

Utilizando las características del VDMA se pueden realizar transferencias de datos en forma independiente del PS. El PS configura e inicializa el VDMA mediante la comunicación AXI4-Lite (conexión en color rojo en la Figura 6-17). Las interfaces AXI (maestro y esclavo, MM2S y S2MM) del VDMA se conectan a un bloque *AXI\_Interconnect* [70] (color verde y amarillo respectivamente en la Figura 6.17), que a su vez las conecta al puerto S\_AXI\_HP0 (*Slave High Performance AXI*) del PS (color naranja de la Figura 6.17). Mediante este puerto se pueden comunicar el PS y el PL, formando un bus de datos de 32 o 64 bits. Para el caso del VDMA la parte PL actúa como maestro y el PS como esclavo [70].

Recordando que la captura de datos se hace por medio del PL, la interfaz AXI4-Stream esclavo, S2MM del VDMA (color rosa de la Figura 6.17), está encargada de recibir los datos del módulo de captura (color rosa en la Figura 6-17). Al recibir los datos el VDMA mapea los datos a la memoria DDR por medio del bus AXI maestro conectado al PS. La lectura de datos para su uso en el PL es por medio de la interfaz AXI4-Stream maestro, MM2S del VDMA (señal marcada en azul en la Figura 6.17).

En la Figura 6.17 se denota con color rojo la conexión encargada de enviar los datos de configuración del VDMA desde el CPU (PS) y comunicarse si es necesario con los IP diseñados para su configuración para el procesamiento de imágenes, por esta razón se pueden observar 2 maestros para la comunicación AXI-Lite. El módulo VDMA se interconecta al puerto M\_AXI\_GP0 (*Master General Purpose AXI*) del PS por medio del IP *AXI\_Interconnect*, (IP

que permite la conexión de un maestro con varios esclavos) formado por un bus de 32 bits adecuado para la comunicación de velocidad media y directa.

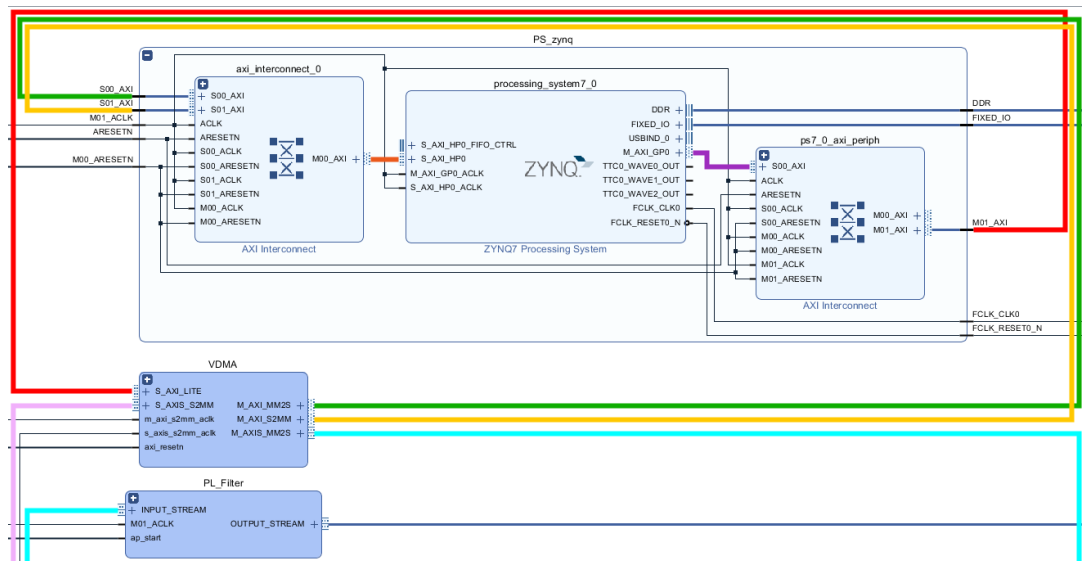


Figura 6.17 Interacción del módulo IP con el PS y PL mediante las Interfaces AXI

El CPU se encarga de configurar e iniciar la transmisión de datos por DMA entre la memoria y los IP desde la aplicación diseñada en el SDK para el CPU (ARM). La configuración de VDMA incluye la dirección de memoria en donde se guardan los datos, siendo útil para acceder a datos de la imagen guardada, así como modificarlos si es necesario desde el ARM. Las direcciones de memoria son proporcionadas por *Vivado IP Integrator*, tras ejecutar la asignación de memoria como se observó en la Fig 2.17.

La configuración de VDMA que se usó para este proyecto se basó en las características de la cámara OV7670, se configuró con la dirección de memoria y las dimensiones de la imagen a recibir, esto se muestra en la parte de código en el ARM como:

```
// dirección de memoria del buffer
unsigned int srcBuffer = (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x100000);
..
/* función en ARM donde API configura el VDMA y empieza la transmisión de datos
sin interrupción. */
Status = run_triple_frame_buffer(&InstancePtr,0, 640, 480, srcBuffer, 100,0);
```

La configuración del IP en la parte PL se muestra en la Figura 6.18.

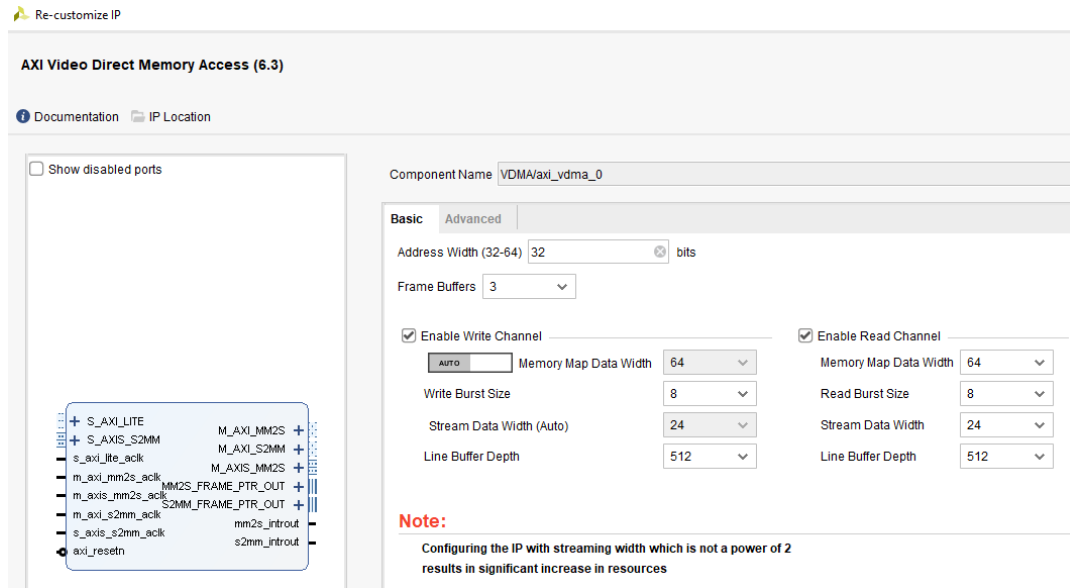


Figura 6.18 Configuración del VDMA.

#### 6.4.2. Adaptación del módulo ov7670\_capture

Para el caso del módulo *ov7670\_capture* se agregó una salida maestra AXI4-Stream MM2S (*Memory map to stream*), por lo tanto, se agregaron las siguientes señales a la entidad de *ov7670\_capture*:

```
entity ov7670_axi_stream_capture_v2 is
  port (
    pclk           : in  std_logic;
    vsync          : in  std_logic;
    href           : in  std_logic;
    d              : in  std_logic_vector (7 downto 0);
    m_axis_tvalid   : out std_logic;
    m_axis_tready   : in  std_logic;
    m_axis_tlast    : out std_logic;
    m_axis_tdata    : out std_logic_vector (23 downto 0 );
    m_axis_tuser    : out std_logic;
    aclk           : out std_logic
  );
end ov7670_axi_stream_capture_v2;
```

Cuando los datos capturados están listos, la señal *m\_axis\_tvalid* se mantiene activa significando datos válidos. Cuando el módulo esclavo está listo envía una señal activa por *s\_axis\_tready* (VDMA) y es captada por la señal *m\_axis\_tready* del módulo de captura. En

cuanto estas señales se activan inicia la transmisión de datos y la transmisión finaliza cuando la cámara no envía datos válidos o el esclavo cancela la comunicación por medio de la señal *ready*. Como se observa la adaptación del módulo para el envío de datos por medio de AXI4-Stream depende mucho de la frecuencia de operación de la cámara que es de 25 MHz y el VDMA puede aceptar esa frecuencia de operación ya que están sincronizados por medio de la señal *aclk* de la cámara y *S\_AXIS\_S2MM* del módulo VDMA permitiendo la captura de datos a esa frecuencia de operación. La cámara envía datos cuando se termina su captura, por lo tanto, la señal *aclk* es la negación de *pclk*.

El módulo del diseño se nombró *ov7670\_axi\_stream\_capture\_v2.vhd* indicando que cuenta con una interfaz AXI4-Stream, los datos siguen siendo captados del mismo modo que se describió en el capítulo 4 agregando una adaptación de RGB444 a RGB888, se muestra en el siguiente código descriptivo:

```
m_axis_tdata <= d_latch(4 downto 0) & d_latch(0) & d_latch(0) &
d_latch(0) & d_latch(10 downto 5) & d_latch(5) & d_latch(5) &
d_latch(15 downto 11) & d_latch(11) & d_latch(11) & d_latch(11);
d_latch(10 downto 5) & d_latch(5) & d_latch(5);
m_axis_tvalid <= we_reg; -- dato valido, sincronizado con pclk
m_axis_tlast <= eol; -- activo durante transmisión de pixeles
m_axis_tuser <= sof; -- activo durante cada salto de cuadro
aclk <= not pclk; -- reloj de sincronización
```

Solo se desliza el dato hacia la izquierda llenando con 0 los datos faltantes. Como se maneja una imagen de grises, también se da la opción de copiar el canal verde (canal con más información) en los tres canales RGB para ser enviados en *m\_axis\_tdata*. El bloque IP de la captura se muestra en la Figura 6.19.

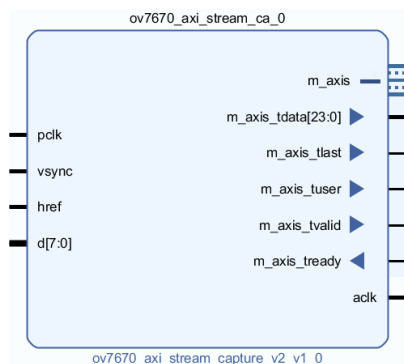


Figura 6.19 Bloque IP para captura con comunicación AXI-Stream.

### 6.4.3. Adaptación del módulo de visualización VGA

Al igual que en el módulo de captura se utiliza la misma metodología para modificar el módulo *vga.vhd*. Se agregaron las señales de comunicación AXI4-Stream para obtener la entidad del módulo VGA;

```
entity AxiStreamVGA is
  Port
    (ACLK          : in  std_logic;-- AXI stream clock
     ARESETN       : in  std_logic;-- reset
     S_AXIS_TVALID: in  std_logic;-- datos validos
     S_AXIS_TDATA  : in  std_logic_vector(24 downto 0);--datos RGB
     S_AXIS_TLAST  : in  std_logic;--
     S_AXIS_TREADY: out std_logic;--
     S_AXIS_TUSER  : in  std_logic;--'1' inicio de la transmisión
     err           : out std_logic;-- '1' indica error
     PIXELCLK      : in  std_logic;-- pixel clock
     VGA_HS        : out std_logic;-- horizontal
     VGA_VS        : out std_logic;-- vertical
     VGA_R         : out std_logic_vector(4 downto 0);
     VGA_G         : out std_logic_vector(4 downto 0);
     VGA_B         : out std_logic_vector(4 downto 0)
    );
end AxiStreamVGA;
```

Se sigue el mismo orden aplicado a las señales en el módulo de captura, solo que ahora para recepción de datos. Otra opción es el uso del IP *AXI4-Stream to Video Out* incluido en Vivado y trabaja en conjunto con el IP *Video Timing Controller* [71], [72], permitiendo la lectura de datos por AXI4-Stream y salida de video con características de transmisión definidas respectivamente. En estos IP se configura la resolución y tiempos de despliegados correspondientes a la salida VGA. Las conexiones entre los IP se definen en los manuales de usuario como se muestra en la Figura 6.20 con la información para su uso en el diseño.

La configuración de los módulos se puede hacer desde el diseño en Vivado como se observa en la Figura. 6.21 y 6.22. Se seleccionó la primera opción por practicidad ya que la resolución del video es estática, esto quiere decir que solo se usa el formato VGA a 640x480 pixeles.

El presente capítulo asume un conocimiento técnico sobre conceptos de diseño digital, arquitectura de computadoras, algoritmos para procesamiento de imágenes, entre otros. Es un capítulo complejo en su estructura y aspectos que detallan la implementación del sistema de procesamiento de imágenes en el *Zynq*. Sin embargo, muestra la gran cantidad de estudio,

análisis y trabajo realizado para llegar a la culminación del sistema desarrollado. Una de las principales contribuciones es el entendimiento en el flujo de datos, donde, se muestra como las señales trabajan en conjunto mediante una máquina de estados síncrona para el envío de datos. Los principales problemas encontrados en este capítulo fue la información detallada demasiado técnica, donde cada uno de los IP utilizados, tienen su propio manual para ayudar a la comprensión de su funcionamiento y como se puede usar en el sistema a diseñar.

Los módulos y sistema diseñados se pueden tomar como base para generar un sistema rápidamente sin una comprensión tan abrupta en el funcionamiento de cada IP debido a que la configuración será menor, esto quiere decir que solo se configuraran parámetros simples como resolución, espacio de color, frecuencias base etc.

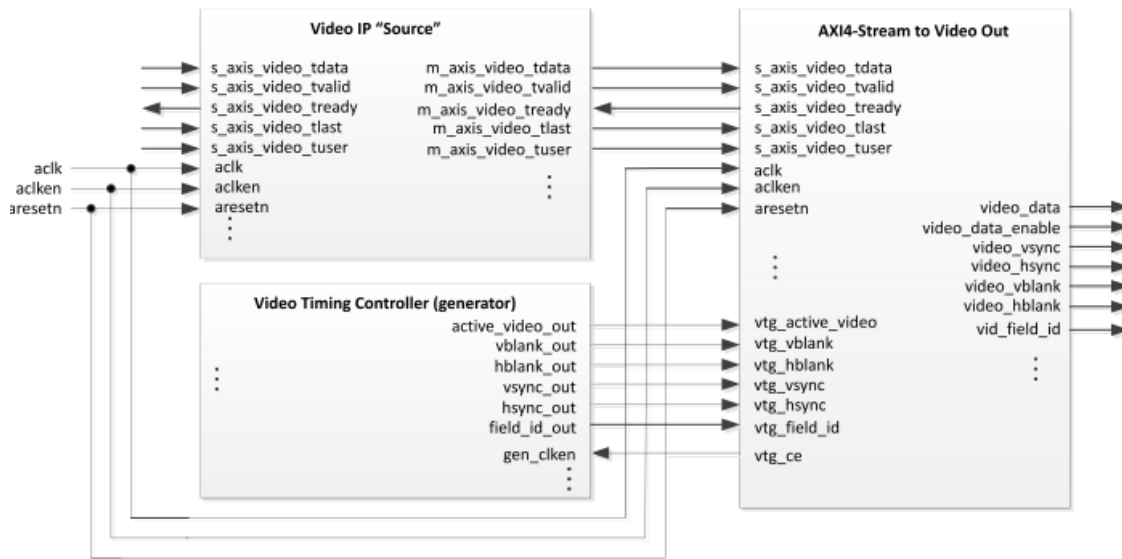


Figura 6.20 Mapa de conexiones entre los IP [71].

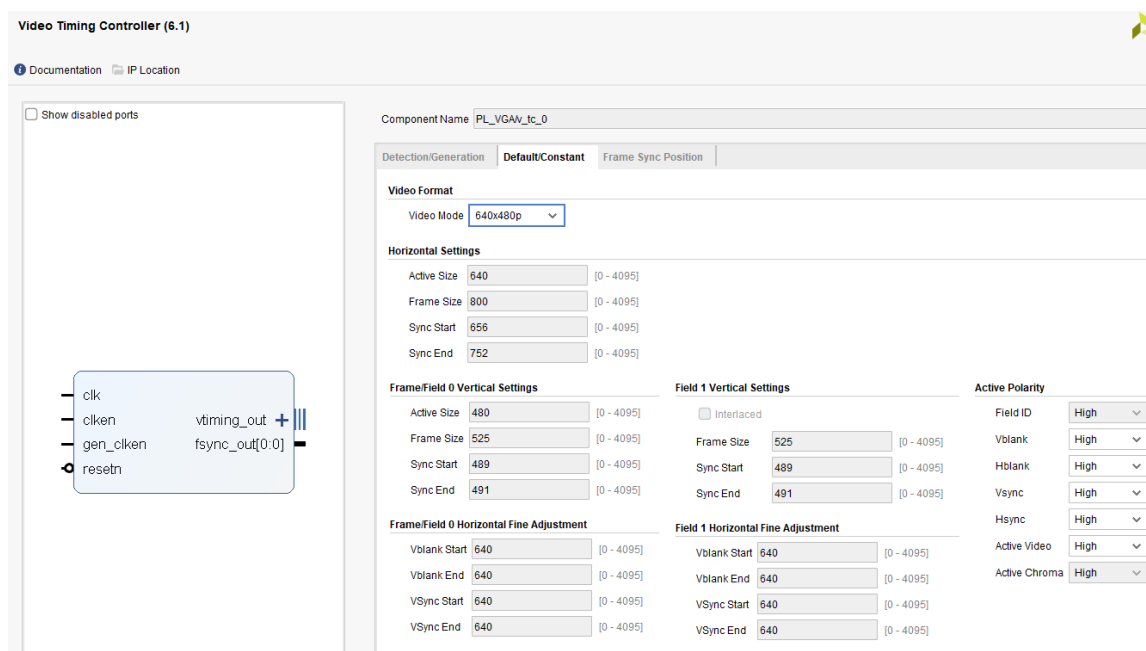


Figura 6.21. Configuración para Video Timing Controller.

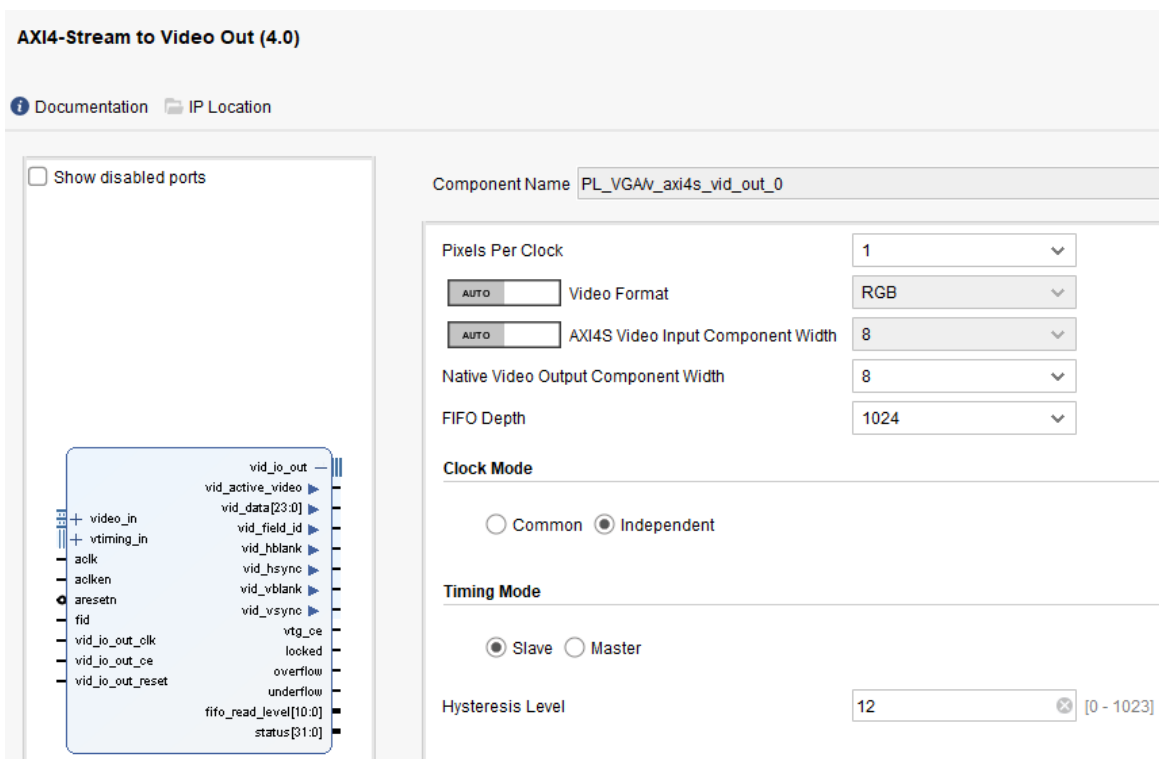


Figura 6.22. Configuración del IP AXI4-Stream to Video Out.

## CAPÍTULO VII

### IMPLEMENTACIÓN DEL SISTEMA Y RESULTADOS

En los capítulos 4 y 6 se describió el desarrollo del hardware para la comunicación y configuración de la cámara de video, periféricos de salida para visualización de la imagen captada y los algoritmos para el procesamiento de imágenes respectivamente. En este capítulo se presenta el sistema de captura trabajando en conjunto con los algoritmos diseñados. Los resultados de los algoritmos son comparados con las funciones similares de Matlab y evaluados con métricas propuestas del desempeño de los algoritmos y el sistema desarrollado.

#### 7.1. Integración del sistema en el SoC FPGA para procesamiento de imágenes

En esta sección se describe la integración de cada uno de los módulos para lograr el sistema SoC FPGA planteado. La Figura 7.1 presenta a grandes rasgos el sistema diseñado, donde se puede observar la integración de los dominios PS-PL del SoC Zynq.

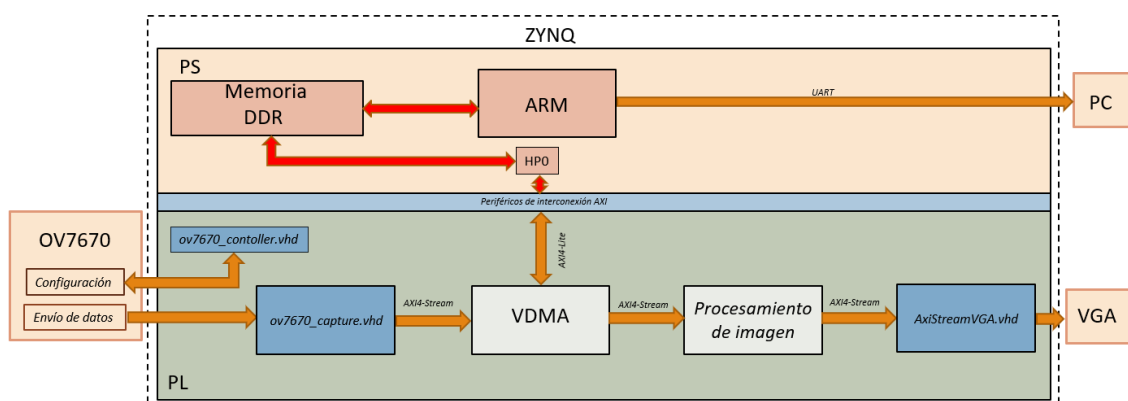


Figura 7.1 Diagrama del sistema diseñado para procesamiento de imágenes en tiempo real.

La comunicación entre la cámara y el sistema se describieron previamente en el capítulo 4 y 6, al igual que la configuración y uso de los periféricos de comunicación AXI. Para la sincronización de las diferentes secciones del sistema fue necesario utilizar tres relojes.

Un reloj de 50 MHz generado desde el PL para la alimentación del controlador de la cámara, un reloj de 25 MHz derivado de la señal PCLK de la cámara y usado para la obtención de datos

y sincronización de transmisión hacia la VDMA. El tercer reloj de sincronización se adquiere de igual manera desde el PL, con la diferencia que inicia su funcionamiento al momento de ejecutar el programa fuente del PS, esto es, para que el VDMA comience a capturar desde el instante que se configura. Por último, la frecuencia de reloj con la que trabaja el ARM es de 666 MHz y la frecuencia de operación de la memoria DDR es de 533 MHz.

Los módulos de procesamiento fueron intercambiados en cada diseño para poder capturar y evaluar el uso de recursos de cada uno de ellos, descartando la opción de utilizar y controlar cada uno de los módulos desde el PS.

Los datos resultantes de los algoritmos son enviados desde el ARM accediendo a la dirección de memoria y capturando los datos para ser enviados por comunicación UART (*Universal Asynchronous Receiver and Transmitter*) a la PC a una velocidad de 115,200 bps.

La velocidad de captura de datos para su envío a la PC fue uno de los problemas resuelto. El problema en el envío de datos fue la velocidad de transmisión, por lo que se decidió utilizar el IP proporcionado por Vivado, *Video Test Pattern Generator* [73]. El IP integra imágenes en su configuración como prueba y puede ser configurado para operar a resoluciones y frecuencias que simulen un sensor de captura, en este caso la cámara OV7670. El IP de video generador de patrones se configura en el PS al igual que el VDMA, tal que se aplicó el código de configuración siguiente:

```
printf("TPG inicialización\r\n");
u32 height,width,status;
status = XV_tpg_IsReady(&ptpg);
printf("Status %u \n\r", (unsigned int) status);
status = XV_tpg_IsIdle(&ptpg);
printf("Status %u \n\r", (unsigned int) status);
XV_tpg_Set_height(&ptpg, (u32)480);
XV_tpg_Set_width(&ptpg, (u32)640);
height = XV_tpg_Get_height(&ptpg);
width = XV_tpg_Get_width(&ptpg);
XV_tpg_Set_colorFormat(&ptpg,XVIDC_CSF_RGB);
XV_tpg_Set_maskId(&ptpg, 0x0);
XV_tpg_Set_motionSpeed(&ptpg, 0x4);
printf("info from tpg %u %u \n\r", (unsigned int)height, (unsigned int)width);
XV_tpg_Set_bckgndId(&ptpg, XTPG_BKGND_CROSS_HATCH); // imagen a transmitir
status = XV_tpg_Get_bckgndId(&ptpg);
printf("Status %x \n\r", (unsigned int) status);
XV_tpg_EnableAutoRestart(&ptpg);
```

```
XV_tpg_Start(&ptpg);
status = XV_tpg_IsIdle(&ptpg);
```

y posteriormente se configura el VDMA como se mencionó en el capítulo 6. El reloj de trabajo del IP se modifica a 25 MHz (este caso) o cualquier frecuencia de operación que se requiera en el diseño. Para la lectura de datos se asignó la dirección de memoria (dirección generada por *Vivado Integrator*) a una variable para poder tener acceso a ella mediante las instrucciones:

```
Pixel_Red = Xil_In8(addr);
Pixel_Green = Xil_In8(addr+1);
Pixel_Blue = Xil_In8(addr+2);
```

donde se lee el primer pixel de la imagen en cada uno de sus componentes de color, recordando que fue la configuración que se eligió en el VDMA para su almacenamiento. La dirección entra en un ciclo *for* de dimensión x-y correspondiente a la resolución de la cámara, 640x480. Después de la lectura de los datos, estos se envían como se mencionó por UART y previamente analizados en Matlab. Se eligió como imagen de análisis, *XTPG\_BKGND\_CROSS\_HATCH*, observada en la Figura 7.2 debido a que en ella se encuentran una amplia combinación de colores en cuadros, además de ser estática.

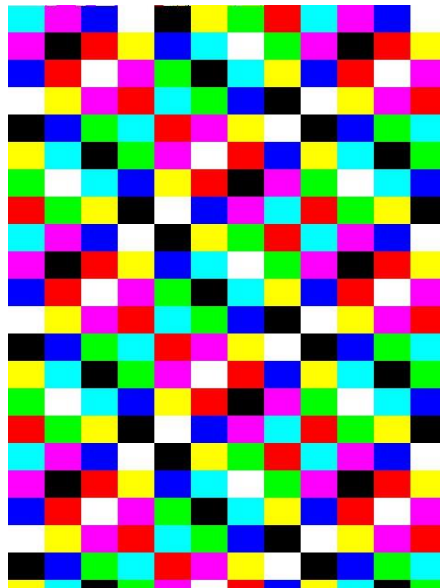


Figura 7.2 Imagen XTPG\_BKGND\_CROSS\_HATCH capturada para análisis en Matlab.

Para obtener los resultados de los algoritmos el sistema se modificó quedando como se muestra en la Figura 7.3. Se muestra que el procesamiento es primero y posteriormente está el

guardado en la VDMA, lo que permite tener acceso a los resultados del algoritmo y ser comprados con los resultados obtenidos en Matlab. Las frecuencias de operación fueron las mismas que en el sistema ilustrado en el diagrama de la Figura 7.1, no obstante, con el nuevo sistema también se llevó a cabo un estrés en la frecuencia de operación de los algoritmos para evaluar el Zynq y saber si es capaz de sintetizar e implementar los algoritmos a frecuencias altas (100 MHz, 200 Mhz), esto para tener una noción de los límites funcionales del chip

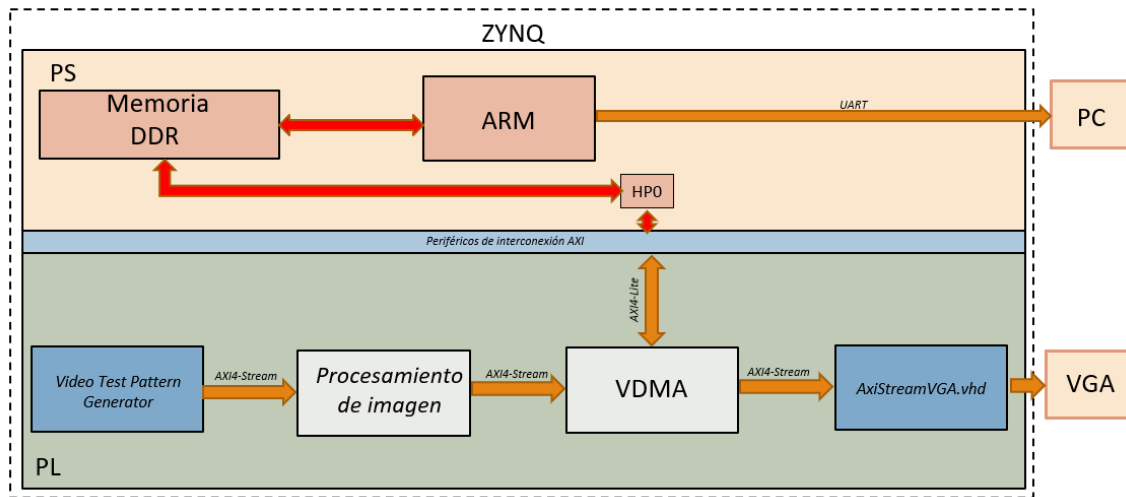


Figura 7.3 Modificación a sistema para evaluación de resultados.

## 7.2. Métricas Utilizadas para Evaluación del Rendimiento

Los criterios tomados en cuenta, cuando se evalúan los resultados de los algoritmos de procesamiento de imágenes digitales, verifican la distorsión que sufre la imagen procesada de salida con una imagen de referencia. En este caso la imagen de referencia serán los resultados del mismo algoritmo utilizando las funciones que Matlab tiene integradas.

La diferencia entre los resultados se puede medir de dos maneras, subjetiva y objetivamente. Cuando un humano evalúa a vista directa las diferencias de los resultados se dice que es de manera subjetiva. Este método no es práctico y es poco empleado, debido a las discrepancias que se pueden generar entre diferentes humanos evaluadores. La manera de evaluar objetivamente utiliza funciones definidas, donde se calcula la diferencia de los resultados considerando los valores de cada pixel de las imágenes. Existen métodos y funciones que miden la diferencia cuantitativamente entre dos imágenes como son: el Error Cuadrático Medio (*Mean*

*Square Error*, *MSE*), Relación de Señal a Ruido de Pico (*PSNR*, *Peak Signal to Noise Ratio*) y Relación de Rendimiento (*PR*, *Performance Ratio*) [74], [75], donde se define *MSE* como:

$$MSE = \frac{1}{N} \sum_{i=1}^N (Iz_i - IM_i)^2 \quad (7.1)$$

*PSNR* como:

$$PSNR = 10 \log_{10} \left( \frac{(2^n - 1)^2}{\sqrt{MSE}} \right) \quad (7.2)$$

donde  $N$  es el número de píxeles en la imagen,  $Iz_i$ , la cual sera la imagen resultante en este caso en el algoritmo implementado en el *Zynq*.  $IM_i$  es la imagen con valores obtenidos con el algoritmo implementado en Matlab y  $n$  es el número de bits que se usan para representar el valor de un píxel. *PR* representa la relación medida y la calculada de bordes verdaderos a falsos, tal que:

$$PR = \frac{\text{verdaderos}(\text{píxeles identificados como verdaderos})}{\text{falsos} + \text{píxeles identificados como no verdaderos}} * 100 \quad (7.2)$$

### 7.3. Resultados

A continuación, se mostrarán los resultados visuales de cada uno de los algoritmos implementados, su comparación con Matlab y recursos utilizados en la implementación final con la integración de todos los módulos.

#### 7.3.1. Detección de bordes en 8 direcciones

Los resultados en recursos consumidos por este algoritmo en la plataforma de desarrollo se muestran en la Figura 7.4. Los recursos mostrados se obtuvieron en el sistema final con un reloj de operación de 25 MHz, obteniendo una tasa de procesamiento de 60 cuadros por segundo, frecuencia a la que opera el monitor. La Figura 7.5 muestra la apreciación visual de los resultados del algoritmo donde se observan todas las variaciones de intensidad ante bordes presentes, la apreciación visual nos ayuda a entender como está funcionando el algoritmo y

cuáles son los bordes que aporta el sistema de visión, los cuales pueden ser usados para un análisis más detallados como una segmentación, pudiendo mostrar 60 cuadros por segundo procesados.

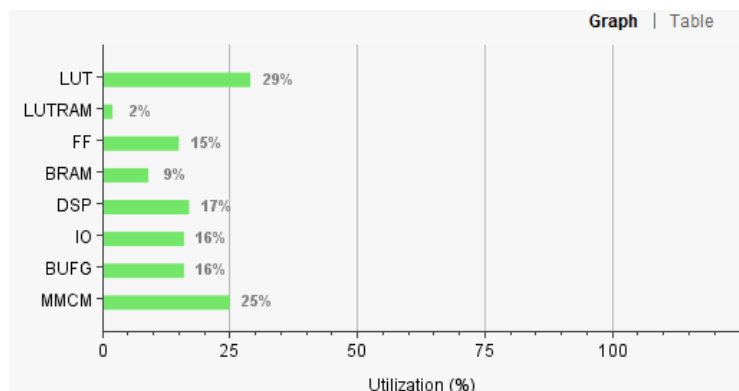


Figura 7.4 Recursos utilizados en la implementación de detección de bordes en 8 direcciones por método de Sobel.



Figura 7.5 Datos captados por la cámara OV7670 y desplegados por VGA a 60 cuadros por segundo.

Los resultados del sistema *Zynq* se compararon contra la salida del mismo algoritmo pero implementado en Matlab, esto para obtener un criterio más objetivo. En la Figura 7.6, se muestra la comparación Matlab donde se utilizó como entrada al sistema la imagen de la Figura 7.2. También, se calcularon las métricas previamente explicadas. Cabe mencionar que se tomó una sección de 100x300 píxeles de la imagen para una mejor visualización en los resultados. La métrica *PR* muestra una tendencia alta, indicando una gran cantidad de píxeles verdaderos

positivos; la distorsión de ruido es mayor a 30 dB considerando una imagen semejante a la comparada [74] y para finalizar la métrica *F1 score* o *F measure* tiene valores entre 0 y 1, indicando con 1 que hay una comparación de imágenes iguales (verdaderos positivos todos los pixeles).

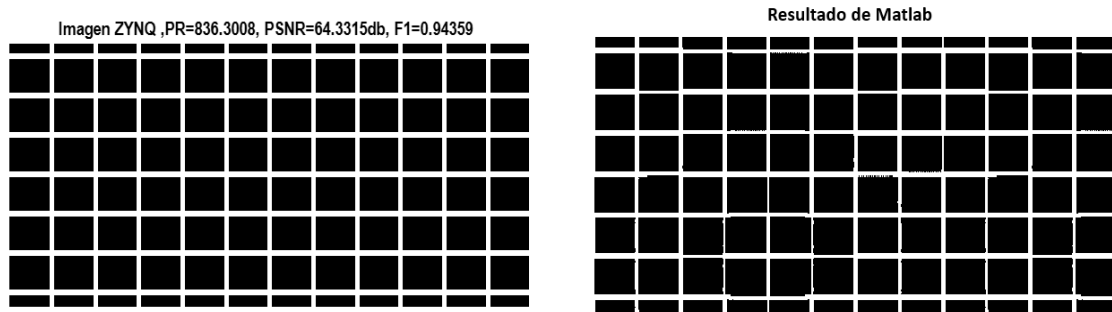


Figura 7.6 Resultados en métricas de evaluación.

En la tabla 7.1 se muestra la comparación de velocidades para el procesamiento de una imagen en el sistema *Zynq* y en Matlab (~, promedio de 30 ejecuciones). Recordando que la visualización de los resultados esta acotada a la frecuencia máxima que soporta el monitor utilizado, aun así, es posible la implementación. El tiempo tomado en Matlab es la conversión de formato RGB a grises, las ocho convoluciones y el cálculo de la magnitud.

Tabla 7.1 Velocidades de procesamiento para el algoritmo de Sobel en ocho direcciones.

| Zynq                 | Matlab (Procesador: Intel Core i7-7700HQ @2.80GHz) |
|----------------------|----------------------------------------------------|
| ~ 11 ms, 25 MHz      | ~30.05 ms                                          |
| ~3.15 ms, 100 MHz    |                                                    |
| ~2.01 ms, 152.39 MHz |                                                    |

### 7.3.2. Detección de esquinas por Harris

Los recursos utilizados en el sistema final para detección de esquinas a una frecuencia de 25 MHz se muestran en la Figura 7.7. Al igual que el algoritmo de Sobel, el resultado se despliega a 60 cuadros por segundo en el monitor VGA. Los resultados de este algoritmo impactaron drásticamente en la utilización de BRAM, esto porque es necesario el almacenar mas información el algoritmo pasado (Sobel 8 direcciones) y el tipo de dato que se maneja es flotante siendo predecible este comportamiento.

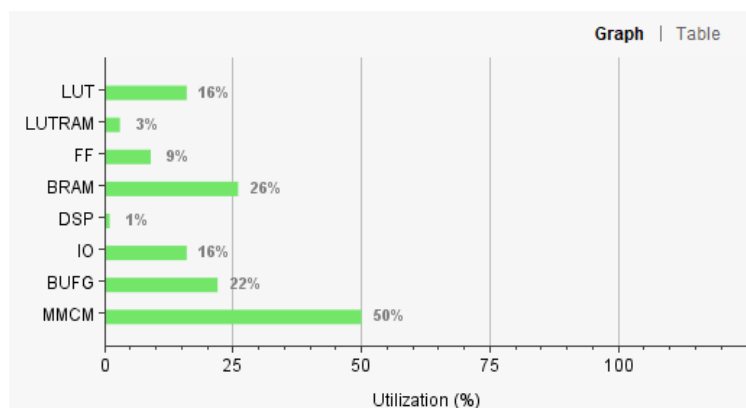


Figura 7.7 Recursos utilizados en el *Zynq* para el detector de esquinas Harris.

La visualización en tiempo real se puede apreciar en la Figura 7.8, donde se detectan esquinas en un tablero de ajedrez y se marcan con puntos de color azul.

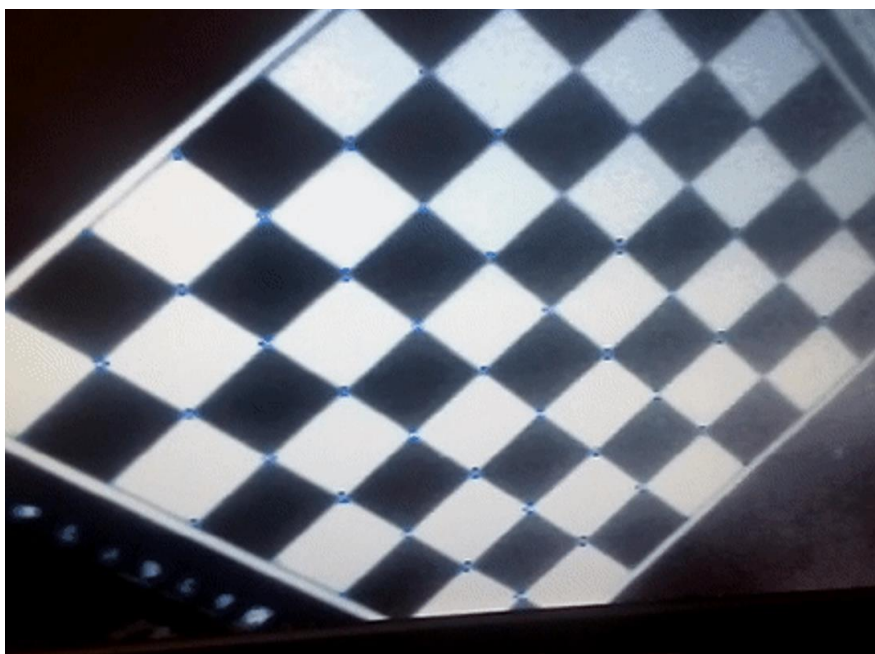


Figura 7.8 Detección de esquinas por Harris en tablero de ajedrez.

La comparación visual del resultado en el *Zynq* y Matlab se observa en la Figura 7.9. Para realizar la comparación, se envió el resultado de las esquinas detectadas en el *Zynq* y se sobrepuso el resultado de la función *detectHarrisFeatures*. Para apreciar mejor el resultado se tomó una sección de 100x300 píxeles de la imagen resultante en la sobreposición.

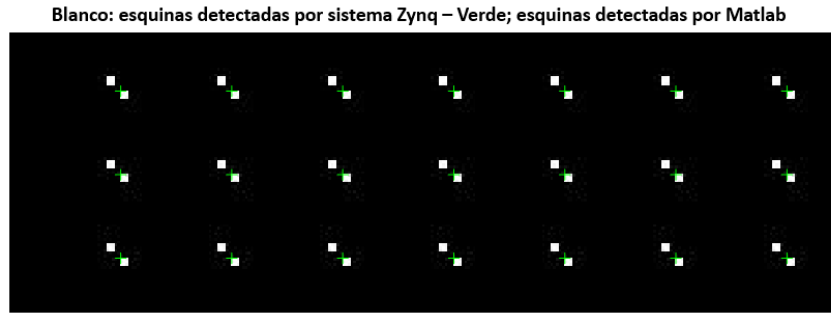


Figura 7.9 Comparación de resultados entre Matlab y el sistema implementado en el *Zynq* para detección de esquinas.

Se puede observar una discrepancia entre ambos resultados, y esto se debe a que la función *detectHarrisFeatures*, encuentra puntos que son máximos locales en las esquinas. Después usa el método *selectStrongest* y selecciona 200 puntos (mismo número de puntos que en la función implementada en el *Zynq*), esos puntos son los de más alta posibilidad de ser esquina. Así, la mecánica es primero encontrar los máximos locales, posteriormente se tienen que ordenar y tomar el “top” 200. También *detectHarrisFeatures* localiza las esquinas con precisión a nivel subpixel, utilizando interpolación lo que provoca la diferencia presentada. Se encontró que se mantiene una constante en la distancia de separación entre los puntos de ambos sistemas como se muestra en la tabla 7.2. Esto demuestra que el resultado del algoritmo implementado en el *Zynq* es aceptable tomando en consideración que no se trabaja a nivel de subpixel.

Tabla 7.2 Diferencia de distancia entre esquinas encontradas en Matlab y el algoritmo Implementado en el *Zynq*.

| Número de esquina Matlab | Posición Matlab |       | Centro de esquina en Zynq |    | Número de esquina Zynq | Diferencia |       |
|--------------------------|-----------------|-------|---------------------------|----|------------------------|------------|-------|
|                          | x               | y     | x                         | y  |                        | x          | y     |
| 1                        | 40.25           | 21.25 | 37                        | 18 | 1                      | 3.25       | 3.25  |
|                          |                 |       | 42                        | 23 | 2                      | -1.75      | -1.75 |
| 2                        | 40.25           | 51.25 | 37                        | 48 | 3                      | 3.25       | 3.25  |
|                          |                 |       | 42                        | 53 | 4                      | -1.75      | -1.75 |
| 3                        | 40.25           | 81.25 | 37                        | 78 | 5                      | 3.25       | 3.25  |
|                          |                 |       | 42                        | 83 | 6                      | -1.75      | -1.75 |
| 4                        | 80.25           | 21.25 | 77                        | 18 | 7                      | 3.25       | 3.25  |
|                          |                 |       | 82                        | 23 | 8                      | -1.75      | -1.75 |

Para finalizar el análisis de este algoritmo implementado en el *Zynq*, se muestran los tiempos de ejecución a diferentes frecuencias (Tabla 7.3) y se comparan con la velocidad de procesamiento de la función en Matlab.

Tabla 7.3 Velocidades de procesamiento para el algoritmo Harris.

| Zynq                | Matlab (Procesador: Intel Core i7-7700HQ @2.80GHz) |
|---------------------|----------------------------------------------------|
| ~ 11.57 ms, 25 MHz  | ~71.794 ms.                                        |
| ~3.20 ms, 100 MHz   |                                                    |
| ~2.23 ms, 157.77MHz |                                                    |

### 7.3.3. Transformada de Hough para detección de líneas

En la implementación de este algoritmo el uso de recursos fue demandante para una detección de 500 píxeles por línea en la función *hls::HoughLines2* dado que los resultados con un número menor de 100 píxeles por línea, el algoritmo demanda un número considerable de memoria en la cual guardar todos los parámetros ( $\rho$ - $\theta$ ) de los píxeles candidatos pertenecientes a una línea. Esto fue debido a que a mayores píxeles para la detección se descartan puntos que se pueden considerar parte de una línea, como en el caso de 100 píxeles y es porque un solo pixel comparte la misma línea un número “n” de píxeles, obteniendo los resultados con 500 píxeles por línea satisfactorios a comparar con la función de Matlab. Si no existiera este umbral se tendría un número infinito de líneas detectadas como se menciona en la sección 3.5 de este documento, por lo tanto, la Figura 7.10 muestra un uso abrupto en recursos con la configuración en la TH.

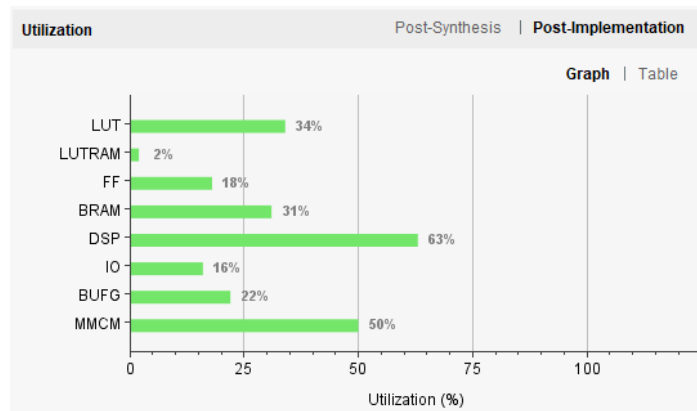


Figura 7.10 Recursos empleados en el sistema para cálculo de la TH en detección de líneas.

Para el análisis de este algoritmo se utilizaron los resultados de la transformada de Hough, esto es, los parámetros  $\rho$ - $\theta$  y se enviaron a Matlab para su comparación. Para tener acceso a

estos datos se utilizó la función *Xil\_In8()*, desde el ARM y mediante la dirección de memoria que asigna Vivado al módulo IP de procesamiento (0x43C0\_0000, para este caso). Los datos fueron transmitidos mediante comunicación UART a la PC.

Los datos fueron analizados con la imagen de la Figura 7.2, misma que se usó en los algoritmos anteriores. Para la comparación, en Matlab se utilizó la función *hough()*, a dicha función se le ingresa una imagen binaria y da como resultado los parámetros de la imagen en el espacio  $(\rho-\theta)$ , después se usa la función *houghpeaks()*, para detectar los puntos que representan una línea (datos en el plano  $(\rho-\theta)$  con mayores puntos de cruce acumulados). Para finalizar se llama a la función *houghlines()* que mapea los picos de puntos detectados a segmentos de línea correspondientes en la imagen, siendo el último paso de este algoritmo. Existirán discrepancias entre los resultados de *Zynq* y Matlab debido a que la función *houghlines()*, utiliza una metodología diferente a la usada en Vivado HLS. La función *houghlines()* en Matlab, utiliza la TH para detección de líneas de manera probabilística, utilizando umbrales para detectar un mínimo de puntos que pueden formar una línea y distancia máxima entre dos píxeles a considerar en la misma línea. Caso contrario la función que es implementada en el *Zynq* solo considera como mínimo número en los acumuladores para la obtención de las líneas y no la distancia entre los puntos considerados como pertenecientes a una línea, no siendo una TH probabilística.

El número de puntos pertenecientes a una línea en el *Zynq* fue de 19,333 donde cada punto cuenta con sus parámetros  $\rho$  y  $\theta$ , donde  $\theta$  está en radianes. La transformada de Hough utilizando Matlab obtuvo un total de 1,599 píxeles candidatos a línea. En la Figura 7.11 se muestran los resultados obtenidos por la detección de líneas en ambos sistemas, donde Matlab obtuvo una detección de cinco líneas y el sistema diseñado en el *Zynq* dos líneas, las cuales son marcadas con flechas de color naranja.

Las condiciones en ambas funciones, tanto de Matlab y en el *Zynq* se trataron de igualar, ambas con 500 puntos, y para Matlab en el segundo parámetro de la función se manejó una distancia de separación entre líneas detectadas de 1, distancia mínima aceptada por la función.

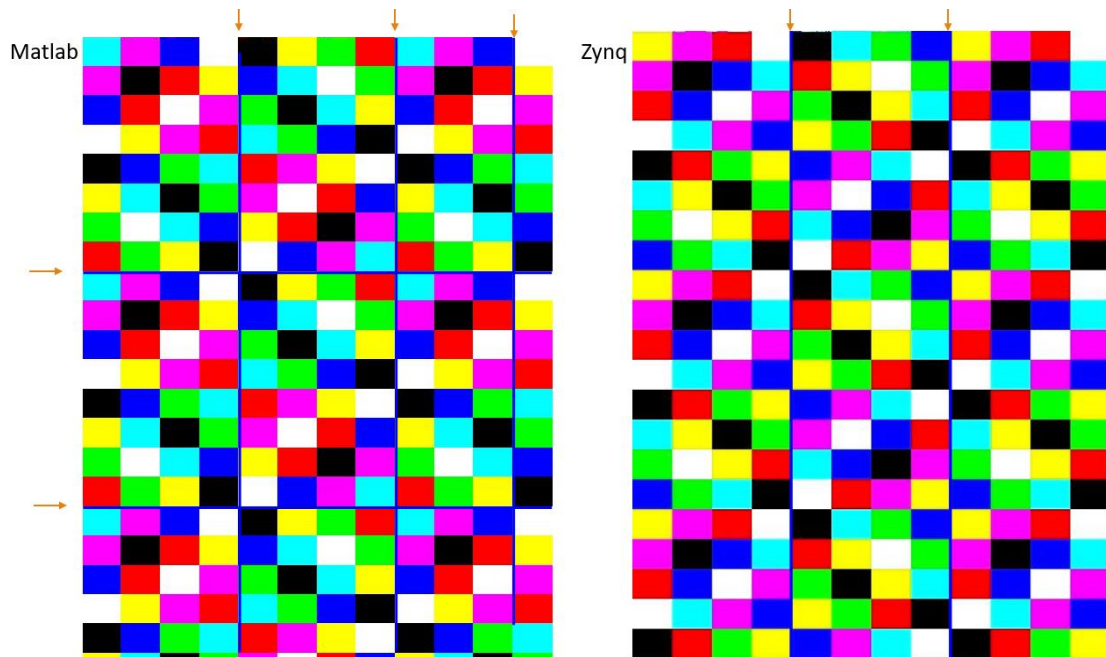


Figura 7.11 Resultados de la TH para detección de líneas.

Para finalizar el análisis del algoritmo, se presentan las velocidades (Tabla 7.4) de procesamiento por imagen para la obtención de los parámetros  $\rho$  y  $\theta$  que son usados en la detección de líneas. Cabe mencionar que los resultados solo se probaron en estas frecuencias y hasta el paso de la obtención de los parámetros de línea en  $\rho$  y  $\theta$  por falta de recursos en el Zynq.

Tabla 7.4 Velocidades de procesamiento para el algoritmo de la TH para detección de líneas.

| Zynq               | Matlab (Procesador: Intel Core i7-7700HQ @2.80GHz) |
|--------------------|----------------------------------------------------|
| ~ 13.60 ms, 25 MHz | ~110.9 ms.                                         |
| ~3.6 ms, 100 MHz   |                                                    |

De los resultados obtenidos en los tres algoritmos evaluados, en velocidad de ejecución se tuvo un rendimiento bastante aceptable aún con el reloj de frecuencia mínima en comparación con las implementaciones en Matlab ya que el FPGA nos permitió la ejecución de tareas en paralelo principalmente la convolución, y el HLS una ejecución del *Pipeline* con *dataflow*, logrando leer y procesar pixeles en un mismo ciclo, en el aspecto de que tan confiables fueron los resultados se pudo mencionar que los resultados comparados con Matlab difieren, pero no lo suficiente para no hacerlos confiables, recordemos que los datos en el Zynq pierden valores

al pasar un flotante a entero. Falta incrementar el número de métricas en las pruebas y lograr una modularización orientada a estas pruebas sin necesidad de modificar el sistema de procesamiento de imágenes original.

## CAPÍTULO VIII

### CONCLUSIONES

En este capítulo se expondrán las diferentes conclusiones a las que se llegaron a lo largo de la realización de este proyecto de tesis. Uno de los primeros puntos a analizar es observar los objetivos planteados y logrados en esta tesis. En el documento se presenta el diseño e implementación y prueba de un sistema digital capaz de adquirir imágenes, aplicarles un algoritmo de procesamiento y desplegar el resultado por medio del periférico VGA a una tasa de 60 cuadros por segundo.

El sistema está basado en la plataforma SoC *Zynq* de Xilinx, donde se aprovecharon los recursos y capacidades del procesamiento en paralelo otorgadas por el FPGA (PL) y el control de los datos en el ARM (PS), ambos integrados en el chip. El reto inicial del proyecto fue la familiarización con un hardware y software de relativa complejidad para el diseño y desarrollo con la plataforma *Zynq* y circuitos periféricos asociados e integrados en la tarjeta *ZedBoard* de Diligent. Se utilizaron herramientas de software “tradicionales” y novedosas para el diseño del hardware, lenguajes de descripción de hardware como VHDL y el sistema HLS en el entorno de desarrollo integrado (IDE) de Vivado, respectivamente.

Se definió el tipo de algoritmos de procesamiento de imágenes a implementar en función del grado de complejidad de éstos. Se trabajó en tres niveles de complejidad, esto es, con algoritmos de procesamiento pixel a pixel o punto a punto de baja complejidad; filtros de ventana de complejidad media como Sobel con ocho direcciones y algoritmos de alta complejidad, detección de esquinas de Harris y Transformada de Hough.

En la validación y evaluación de los diseños se derivaron opciones de modificación a los mismos para lograr obtener métricas de rendimiento que permitirían la comparación en términos de tiempo de ejecución con los mismos algoritmos implementados en Matlab.

Existieron muchos detalles de prueba y error a lo largo del desarrollo y obviamente derivaron en aprendizaje de las capacidades y uso tanto en software como en el hardware. Se comentan

algunos de los principales elementos que tuvieron un impacto importante en el diseño final del sistema de procesamiento.

Cuando se analizó el sistema desarrollado para la realización de los algoritmos de procesamiento de imágenes en *software* (en el ARM), se detectaron problemas de velocidad al momento de acceder a la memoria donde se guardó la imagen, así como sobrescribir el resultado en esta misma. Los algoritmos de procesamiento fueron instanciados en *hardware*, FPGA, dejando el ARM a la parte de control y se logró contar con un diseño que involucra la interacción del FPGA con el ARM en el *Zynq*. El sistema final es capaz de procesar imágenes a tasas considerables y obtener una visualización en tiempo real.

En relación con las metas planteadas y logradas se puede concluir:

- Se cuenta con una plataforma SoC-FPGA para el procesamiento digital de imágenes con visualización en tiempo real la cual fue probada con los algoritmos demandantes computacionalmente.
- Se desarrollaron e implementaron módulos para la adquisición de datos y configuración de la cámara OV7670. Al módulo de obtención de datos se le adaptó un canal de comunicación AXI4-Stream para el envío de datos a la memoria DDR por medio del VDMA.
- Se obtuvieron metodologías de diseño en Vivado HLS, en el flujo de implementación de los algoritmos de procesamiento de imágenes. Se demostró la posibilidad de realizar un desarrollo rápido en tiempo utilizando esta herramienta en lugar de solo VHDL. No obstante, las funciones otorgadas por esta herramienta, así como por las funciones de la librería OpenCV sintetizadas en hardware, no están completamente optimizadas. Por ello, se optó por estudiar el flujo de diseño para la sinterización de las funciones, mostrando un conocimiento profundo en lenguajes de programación de alto nivel como C/C++, para la utilización de Vivado HLS.
- Se logró la sintetización de módulos IP desde Vivado HLS los cuales se pueden adaptar a las frecuencias requeridas por el hardware a utilizar. Si bien se puede usar Vivado HLS para la sinterización de código C/C++ a FPGA como C a cualquier CPU, el proceso de sintetizar

comprendió el aprender directivas, procedimiento y sintaxis para su uso. Así, se logró sacar provecho a las directivas existentes en Vivado HLS para el tipo de compilación del algoritmo, no obstante, existen en HLS elementos adicionales que no se utilizaron y que podrían ser auxiliares en el desarrollo de los diseños. Vivado HLS es una herramienta con muchísimas opciones de trabajo y definitivamente faltó el conocer algunas de ellas.

- El diseño final implementado en base de la integración FPGA-CPU (dominios PL-PS) otorgada en el *Zynq* aprovecha cada una de las cualidades que presenta la arquitectura. La integración de ambos dominios arquitecturas requirió de conocimiento en las interfaces o canales de comunicación entre PS y PL del *Zynq* obteniendo una plantilla de comunicación AXI4 en VHDL.
- Se evaluó el sistema final obteniendo resultados comparables en tiempo de ejecución con los ofrecidos por Matlab, destacando la velocidad de procesamiento por imagen que puede otorgar el sistema en los algoritmos de Harris y Sobel en 8 direcciones.

Uno de los problemas en la realización de este proyecto fue la correcta sincronización de todos los módulos IP que conforman al sistema, problema corregido mediante el uso de las comunicaciones AXI4 y el uso de la IP VDMA, lo cual permitió la unión de módulos con diferentes frecuencias de operación.

En el caso de estudio de la Transformada de Hough (TH), entre los problemas que se tuvieron fue el uso de recursos, si bien en [76] se hace alusión a la gran cantidad de recursos necesarios en la TH, Xilinx muestra en [67] el uso de la TH para detección de líneas mostrando una gran cantidad de recursos utilizados en la tarjeta de evaluación ZCU102 [77]. Queda para un estudio más profundo el manejo de recursos para esta aplicación, ya sea optimizando la función en Vivado HLS con el uso de directivas exclusivas o un manejo de la función en VHDL puro, que por cuestiones de tiempo no fue posible.

En cuestión de los resultados del sistema, se obtuvieron los esperados ya que al ser sintetizadas las funciones Vivado HLS, se tiene un sistema cuyo diseño es óptimo en el uso de recursos y obtención de datos faltantes debido a los cambios en los tipos de datos en las funciones (flotante-entero). Sin embargo, se demostró que los algoritmos pueden ser

implementados en aplicaciones que demanden una respuesta en tiempo real considerablemente rápida de hasta 300 cuadros por segundo.

### **8.1. Propuesta de trabajo futuro**

En cuanto a los trabajos que puede abrir esta investigación, se propone el expandir la resolución de captura de video, esto debido a que se trabajó con un sensor de bajo coste y por ende la resolución es pequeña. El trabajar con un sensor que brinde una resolución más alta permitiría evaluar los algoritmos y observar el comportamiento del chip *Zynq* ante demandas de datos mayores. Esto implicaría usar frecuencias altas en el sistema, las cuales estresarán al sistema obteniendo cuellos de botella en las transmisiones debido a la demanda de datos a procesar en instantes de tiempo más cortos.

El usar una resolución más alta en el sistema ya sea incursionar hasta los 4K (resolución  $4096 \times 2160$  pixeles), surgiría la necesidad de cambiar la interfaz VGA por HDMI, permitiendo una visualización en frecuencias de procesamiento altas las cuales nos son posibles en VGA, de igual manera el incursionar en este protocolo de comunicación abrirá más el conocimiento en las visualizaciones de video de alta calidad.

Durante el desarrollo de la investigación se logró demostrar y observar las grandes capacidades de Vivado HLS, lo cual sería interesante incursionar más. Observar el comportamiento de todas las directivas que nos ofrece e incluso proponer o modificar algunas de ellas y detallar el comportamiento sería de gran ayuda.

Con lo anteriormente mencionado se propone realizar un sistema de visión para una tarea en específico como por ejemplo, vigilancia, segmentación y seguimiento de objetos, calidad de video, etc., e incluso incursionar con el aprendizaje automático en los con *Zynq*. Xilinx es una empresa en conocimiento de alto nivel en FPGA y provee de IP especializado para estas tareas. Investigar sobre las metodologías de aprendizaje automático desarrolladas en lenguaje descriptivo ayudaría a tener un incremento en velocidad de procesamiento así como un manejo de datos a alta velocidad [78].

## REFERENCIAS

- [1] Z. Xiangxi, Z. Yonghui, Z. Shuaiyan and Z. Jian, "*FPGA implementation of edge detection for Sobel operator in eight directions*," *2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, Chengdu, 2018, pp. 520-523.
- [2] V. Kamatchi Sundari, M. Manikandan, P. Prakash, "*FPGA Implementation of Sobel Edge Detector*", *International Journal of Advances in Science and Technology (IJAST)*, 15 February 2014, pp. 255-259.
- [3] S. Yahia, "*HW/SW Co-design for FPGA based Video Processing Platform*," *Arch. Des Sci.*, 2012, vol. 65, no. 12, pp. 504–515.
- [4] I. Yasri, N. H. Hamid and V. V. Yap, "*Performance analysis of FPGA based Sobel edge detection operator*," *2008 International Conference on Electronic Design*, Penang, 2008, pp. 1-4.
- [5] Ankush R. Bhagat, Swati R. Dixit, Dr. A.Y. Deshmukh "*VHDL based Sobel Edge Detection*" *International Journal of Engineering Research and General Science*, vol 3, Issue 1, 2015.
- [6] E. A. Perdomo Hourné, L. M. Garcés-Socarrás, and A. J. Cabrera Sarmiento, "*Diseño de bloques para el procesamiento de imágenes en lenguaje de descripción de hardware*," *Rev. Ing. Electrónica, Automática y Comun.*, vol. XXXIV, no. 2/2013, pp. 9–18, 2013.
- [7] M. Balaji and S. A. Christe, "*FPGA implementation of various image processing algorithms using xilinx system generator*," *Smart Innov. Syst. Technol.*, vol. 32, no. 4, pp. 59–68, 2015.
- [8] G. Kiefer, M. Vahl, J. Sarcher and M. Schaeferling, "*A configurable architecture for the generalized hough transform applied to the analysis of huge aerial images and to traffic sign detection*," *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, 2016, pp. 1-7.
- [9] H. H. Mouane et al., "*FPGA-based system for the road signs detection*," *2017 International Conference on Electrical and Information Technologies (ICEIT)*, Rabat, 2017, pp. 1-6.
- [10] J. Zhao, B. Thörnberg, Y. Shi and A. Hashemi, "*Color segmentation on FPGA using*

- minimum distance classifier for automatic road sign detection," 2012 IEEE International Conference on Imaging Systems and Techniques Proceedings, Manchester, 2012, pp. 516-521.*
- [11] M. Russell and S. Fischhaber, "OpenCV based road sign recognition on Zynq," *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, Bochum, 2013, pp. 596-601.
  - [12] W. Farhat, H. Faiedh, C. Souani and K. Besbes, "Real-time hardware/software co-design of a traffic sign recognition system using Zynq FPGA," *2016 11th International Design & Test Symposium (IDT)*, Hammamet, 2016, pp. 302-307.
  - [13] Y. Han and E. Oruklu, "Real-time traffic sign recognition based on Zynq FPGA and ARM SoCs," *IEEE International Conference on Electro/Information Technology*, Milwaukee, WI, 2014, pp. 373-376.
  - [14] F. Schwiegelshohn and M. Hübner, "Design of an attention detection system on the Zynq-7000 SoC," *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, Cancun, 2014, pp. 1-6.
  - [15] F. Siddiqui, S. Amiri, U. I. Minhas, T. Deng, R. Woods, K. Rafferty and D. Crookes, "FPGA-Based Processor Acceleration for Image Processing Applications," *Journal of Imaging*, 2019, vol. 5, pp. 16.
  - [16] Aiman Badawi and Muhammad Bilal, "High-Level Synthesis of Online K-Means Clustering Hardware for a Real-Time Image Processing Pipeline", *J. Imaging* 2019, vol. 5, pp. 38.
  - [17] A. K. Gupta, D. J. Bora, "A Novel Color Image Segmentation Approach Based On K-Means Clustering with Proper Determination of the Number of Clusters and Suitable Distance Metric", *International Journal of Computer Science & Engineering Technology (IJCSET)*, Vol. 7 No. 09 Sep 2016, pp. 395-409.
  - [18] M. S. A. Christe, M. Vignesh, and A. Kandaswamy, "An Efficient Fpga Implementation of Mri Image Filtering and Tumour Characterization Using Xilinx System Generator," *Int. J. VLSI Des. Commun. Syst.*, 2011, vol. 2, no. 4, pp. 95–109.
  - [19] W. Liu, H. Chen, and L. Ma, "Moving object detection and tracking based on Zynq FPGA and ARM SoC," *IET Conf. Publ.*, 2015, no. CP677, pp. 2–5.

- 
- [20] M. A. kumar, “*Design and Implementation of Zynq-Based Reconfigurable System for Jpeg 2000 Compression*,” *Int. J. Res. Appl. Sci. Eng. Technol.*, 2017, vol. V, no. XI, pp. 402–407.
  - [21] L. Cavigelli, M. Magno, and L. Benini, “*Accelerating real-time embedded scene labeling with convolutional networks*,” *Proc. - Des. Autom. Conf.*, 2015-July.
  - [22] S. I. Venieris, A. Kouris, and C. S. Bouganis, “*Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions*,” *ACM Comput. Surv.*, vol. 51, no. 3, 2018.
  - [23] N. Sugimoto et al., “*Trax solver on Zynq with Deep Q-Network*,” *2015 International Conference on Field Programmable Technology (FPT)*, Queenstown, 2015, pp. 272-275 .
  - [24] G. Hegde, Siddhartha, N. Ramasamy, V. Buddha and N. Kapre, “*Evaluating Embedded FPGA Accelerators for Deep Learning Applications*,” *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Washington, DC, 2016, pp. 25-25.
  - [25] J. Silva, V. Sklyarov and I. Skliarova, “*Comparison of On-chip Communications in Zynq-7000 All Programmable Systems-on-Chip*,” in *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 31-34, March 2015, .
  - [26] H. Amano, *Design and Structures of FPGAs*, 1st ed. Springer, 2018.
  - [27] G. L. Moss, Ronald J. Tocci, Neal S. Widmer, “*Arquitecturas de los dispositivos lógicos programables*,” *Sistemas digitales : principios y aplicaciones*, 10ma Edici., Pearson, Ed. México, 2007, pp. 869–875.
  - [28] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014.
  - [29] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, “*Zynq Book*,” *Aging (Albany. NY)*, vol. 7, no. 11, pp. 956–963, 2015.
  - [30] N. Eastland, “*FPGA – Configurable Logic Block*,” 2015. [Online]. Available: <https://blog.digilentinc.com/fpga-configurable-logic-block/>.
  - [31] J. Rajewski, “*How does an FPGA work?*,” 2018. [Online]. Available:

- <https://alchitry.com/blogs/tutorials/how-does-an-fpga-work>.
- [32] A. Moore, *FPGAs For Dummies*. Hoboken, New Jersey: John Wiley & Sons, 2017.
  - [33] M. Véstias and H. Neto, "Trends of CPU, GPU and FPGA for high-performance computing," *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Munich, 2014, pp. 1-6 .
  - [34] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with GPUs and FPGAs," *Symp. Appl. Specif. Process. SASP 2008*, pp. 101–107, 2008.
  - [35] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of FPGA, GPU and CPU in image processing," *FPL 09 19th Int. Conf. F. Program. Log. Appl.*, pp. 126–131, 2009.
  - [36] Xilinx, "SDAccel Methodology Guide," 2019. [Online]. Available: [https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2018\\_3/ug1346-sdaccel-methodology-guide.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1346-sdaccel-methodology-guide.pdf).
  - [37] S. Ma, X. Shi, and D. Andrews, "Parallelizing maximum likelihood classification (MLC) for supervised image classification by pipelined thread approach through high-level synthesis (HLS) on FPGA cluster," *Big Earth Data*, vol. 2, no. 2, pp. 144–158, 2018.
  - [38] Arm, "AMBA: The Standard for On-Chip Communication," 2018. [Online]. Available: <https://www.arm.com/products/silicon-ip-system/embedded-system-design/amba-specifications>.
  - [39] S. Ramagond, S. Yellampalli, and C. Kanagasabapathi, "A review and analysis of communication logic between PL and PS in ZYNQ AP SoC," *Proc. 2017 Int. Conf. Smart Technol. Smart Nation, SmartTechCon 2017*, pp. 946–951, 2018.
  - [40] Xilinx and Inc, "AXI4-Stream Infrastructure IP Suite v3.0 LogiCORE IP Product Guide Vivado Design Suite," 2018.
  - [41] A. X. I. Reference, V. Axi, and R. Guide, "Vivado AXI Reference Vivado Design Suite AXI Reference Guide," vol. 1037, 2017.
  - [42] ZedBoard, "ZedBoard v2.2," no. January, 2014 [Online]. Available: <http://zedboard.org/support/documentation/1521>.

- [43] Omnivision, “Ov7670/Ov7171 Cmos Vga,” Converter, pp. 1–32, 2004.
- [44] M. Thompson, R. C. Gonzalez, and P. Wintz, “*Digital Image Processing*,” vol. 14, no. 3, p. 256, 1981.
- [45] Harris C. and Stephens M., “A Combined Corner and Edge Detector,” *Proc. Fourth Alvey Vision Conf.*, vol. 15, pp. 147-151, 1988.
- [46] H. P. Moravec, “*Visual Mapping by a Robot Rover*,” *Proc. 6th Int. Jt. Conf. Artif. Intell.*, vol. 1, pp. 598–600, 1979.
- [47] J. B. Ryu, C. G. Lee, and H. H. Park, “*Formula for Harris corner detector*,” *Electron. Lett.*, vol. 47, no. 3, pp. 180–181, 2011.
- [48] D. A. Lyon, *Image Processing in Java*. Upper Saddle River, NJ: Prentice Hall PTR, 1999.
- [49] OpenCV, “*Hough Line Transform*,” 2018. [Online]. Available: [https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/hough\\_lines/hough\\_lines.html](https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/hough_lines/hough_lines.html).
- [50] Omnivision, “*Serial Camera Control Bus Functional Specification*,” Control, no. March, pp. 0–18, 2002.
- [51] M. Field, “*ZedBoard Ov7670*,” 2013. [Online]. Available: [http://hamsterworks.co.nz/mediawiki/index.php/ZedBoard\\_OV7670](http://hamsterworks.co.nz/mediawiki/index.php/ZedBoard_OV7670).
- [52] Xilinx, “*Block Memory Generator v8.3*,” Data Sheet, pp. 1–129, 2017.
- [53] Florent, “*Create a VGA controller for the ZedBoard*,” 2017. [Online]. Available: <http://blog.dev-flow.com/en/11-create-a-vga-controller-for-the-ZedBoard/>.
- [54] J. SriVarshini and V. NarasimhaNayak, “*Implementation of video-processing and control on a Zynq soc platform*,” *Int. J. Eng. Technol.*, vol. 8, no. 1, pp. 274–279, 2016.
- [55] Xilinx, “*Designing Protocol Processing Systems with Vivado High-Level Synthesis*,” *Xapp1209*, vol. 1209, pp. 1–24, 2014.
- [56] J. de F. Licht, M. Besta, S. Meierhans, and T. Hoefler, “*Transformations of High-Level Synthesis Codes for High-Performance Computing*,” pp. 1–12, 2018.
- [57] G. Martin and G. Smith, “*High-Level Synthesis: Past, Present, and Future*,” in *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 18-25, July-Aug. 2009
- [58] B. C. Schafer and Z. Wang, “*High-Level Synthesis Design Space Exploration: Past*,

- Present and Future,*” *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 11, no. 4, pp. 1–12.
- [59] Xilinx Inc., “*Vivado Design Suite User Guide: High-Level Synthesis,*” Ug 908, 2015. [Online]. Available: [http://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2015\\_4/ug903-vivado-using-constraints.pdf](http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug903-vivado-using-constraints.pdf).
- [60] X. Inc., “*HLS Pragmas.*” [Online]. Available: [https://www.xilinx.com/html\\_docs/xilinx2018\\_1/sdsoc\\_doc/ijd1517252127848-1.html](https://www.xilinx.com/html_docs/xilinx2018_1/sdsoc_doc/ijd1517252127848-1.html).
- [61] A. Allain, “*pragma c and c++ syntax reference.*” [Online]. Available: <https://www.cprogramming.com/reference/preprocessor/pragma.html>.
- [62] Xilinx, “*Vivado HLS Optimization Methodology Guide 2018.1,*” *Xilinx.Com*, vol. 1270, pp. 1–131, 2018.
- [63] X. Inc., “*Loop Pipelining and Loop Unrolling.*” [Online]. Available: [https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2015\\_2/sdsoc\\_doc/totpics/calling-coding-guidelines/concept\\_pipelining\\_loop\\_unrolling.html](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/totpics/calling-coding-guidelines/concept_pipelining_loop_unrolling.html).
- [64] X. Inc., “*Pragma HLS loop\_flatten.*” [Online]. Available: [https://www.xilinx.com/html\\_docs/xilinx2018\\_1/sdsoc\\_doc/hid1517254361170.html](https://www.xilinx.com/html_docs/xilinx2018_1/sdsoc_doc/hid1517254361170.html).
- [65] Xilinx, “*Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Librarie,*” *Xilinx Wiki*, vol. 1167, p. 1, 2013.
- [66] OpenCV, “*Introducción a OpenCV,*” 2017. [Online]. Available: <https://docs.opencv.org/3.3.0/d1/dfb/intro.html>.
- [67] X. Inc., “*Xilinx openCV User Guide (v2018.2),*” vol. 1233, pp. 1–302, 2018.
- [68] Xilinx inc., “*HLS Video Library,*” 2018. [Online]. Available: <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841665/HLS+Video+Library>.
- [69] L. Ip and P. Guide, “*AXI Video Direct Table of Contents,*” Pg430, pp. 1–60, 2016.
- [70] Xilinx, “*AXI Interconnect v2.1 LogiCORE IP Product Guide,*” pp. 1–171, 2017.
- [71] Xilinx Inc., “*Video Timing Controller v6.1 LogiCORE,*” no. may. pp. 1–90, 2019.
- [72] V. D. Suite, “*AXI4-Stream to Video Table of Contents,*” 2017.
- [73] X. Inc., “*Video Test Pattern Generator v8.0,*” pp. 2–4, 2019.
- [74] P. Symes, *Video Compression Demystified*, Pap/Com. McGraw-Hill Professional, 2001.

- [75] P. A. Khaire, “*A Fuzzy Set Approach for Edge Detection*,” *Int. J. Image Process.*, no. 6, pp. 403–412, 2012.
- [76] D. G. Bailey, *Design for Embedded Image Processing on FPGAs*. 2011.
- [77] Xilinx, “*ZCU102 Evaluation*,” vol. 1182, pp. 1–120, 2018.
- [78] I. Xilinx, “*AI Inference Acceleration*,” 2020. [Online]. Available: <https://www.xilinx.com/applications/megatrends/machine-learning.html>.