INSTITUTO TECNOLÓGICO DE LA PAZ
DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN
MAESTRÍA EN SISTEMAS COMPUTACIONALES

IMPLEMENTACIÓN DE UN SISTEMA DE VISIÓN ESTÉREO EN UN DISPOSITIVO FPGA PARA SU USO EN UN SISTEMA AUTÓNOMO DE MEDICIÓN DE PARÁMETROS OCEÁNICOS

QUE PARA OBTENER EL GRADO DE
MAESTRO EN SISTEMAS COMPUTACIONALES

PRESENTA:
MARIO ALBERTO CASTRO MORGAN

DIRECTORES DE TESIS:
DR. ISRAEL MARCOS SANTILLAN MÉNDEZ
DR. JORGE DOMINGO MENDIOLA SANTIBAÑEZ

LA PAZ, BAJA CALIFORNIA SUR, MÉXICO, ENERO 2023.

DICTAMEN DEL COMITÉ TUTORIAL

La Paz, B.C.S., **20/ENERO/ 2023**

JUDITH GUADALUPE MARTÍNEZ TIRADO,
JEFA DE LA DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN,
P R E S E N T E.

Por medio del presente, enviamos a usted dictamen del Comité Tutorial de tesis para la obtención del grado de Maestro, con los siguientes datos generales:

No. de Control M21310002	Nombre MARIO ALBERTO CASTRO MORGAN
Maestría en:	SISTEMAS COMPUTACIONALES
Título de la tesis: IMPLEMENTACIÓN DE UN SISTEMA DE VISIÓN ESTÉREO PARA SU USO EN UN SISTEMA AUTÓNOMO DE MEDICIÓN DE PARÁMETROS OCEÁNICOS	
DICTAMEN: Se autoriza el trabajo de investigación, en virtud de que realizó las correcciones correspondientes conforme a las observaciones planteadas por este Comité Tutorial.	

A t e n t a m e n t e .
El Comité Tutorial

MC. JORGE ENRIQUE LUNA TAYLOR

DR. SAÚL MARTÍNEZ DÍAZ

DR. ISRAEL MARCOS SANTILLÁN MÉNDEZ

DR. JORGE DOMINGO MENDIOLA SANTIBAÑEZ

c.c.p. Coordinador de la Maestría.
c.c.p. Departamento de Servicios Escolares.
c.c.p. Estudiante.

Boulevard Forjadores de B.C.S. #4720, Col. 8 de Octubre 1ra Sección, C.P. 23080, La Paz, B.C.S. Tel. (612) 12 10424 e-mail: depi_paz@tecnm.mx | lapaz.tecnm.mx

La Paz, B.C.S., 24/ ENERO /2023

DEPI_MSC/025/2023

ASUNTO: Autorización de impresión

C. MARIO ALBERTO CASTRO MORGAN,
ESTUDIANTE DE LA MAESTRÍA EN
SISTEMAS COMPUTACIONALES,
P R E S E N T E .

Con base en el dictamen de aprobación emitido por el Comité Tutorial de la Tesis denominada: **"IMPLEMENTACIÓN DE UN SISTEMA DE VISIÓN ESTÉREO PARA SU USO EN UN SISTEMA AUTÓNOMO DE MEDICIÓN DE PARÁMETROS OCEÁNICOS"**, mediante la opción de tesis (Proyectos de Investigación), entregado por usted para su análisis, le informamos que se **AUTORIZA** la impresión.

A T E N T A M E N T E
Excelencia en Educación Tecnológica®

EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA
TECNOLÓGICO NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE LA PAZ
DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

JUDITH GUADALUPE MARTÍNEZ TIRADO,
JEFA DE LA DIV. DE ESTUDIOS DE POSGRADO E INV.

c.c.p. Depto. de Servicios Escolares
c.c.p. Archivo.

*JGMT/icl**

Dedicatoria

Mi madre Perla, que siempre ha hecho lo mejor que puede para que yo y mis hermanos logremos salir adelante.

Agradecimientos

A todas aquellas personas y amigos que estuvieron conmigo durante el desarrollo de este proyecto y cuando más los necesité.

Resumen

En la presente investigación se busca la implementación de un sistema de visión estéreo en un dispositivo de arreglo de compuertas lógicas programables en campo (FPGA por sus siglas en inglés) SoC Xilinx Zynq-7000 para su uso en la navegación de vehículos autónomos para la recolección de parámetros oceanográficos. El diseño se dividió en dos partes fundamentales: la primera tiene el objetivo de hacer la carga y el preprocesado de los resultados de la correlación; para propósito de esta sección los algoritmos se implementan en la parte del sistema programable. En la segunda parte, se diseña la arquitectura para implementar los algoritmos de correlación en la lógica programable del FPGA debido a que tiene el mayor coste computacional. Para hacer las pruebas, se emplearon imágenes reales obtenidas de bases de datos y se comparan con resultados de una PC logrando una mayor velocidad de computo con una confidencia del 60 % empleando imágenes de 320×240. Este primer avance puede servir para la creación de un sistema de navegación mas completo.

Abstract

This research focuses in the implementation of a stereo vision system on a Xilinx Zynq-7000 SoC field programmable gate array (FPGA) device for use in autonomous vehicle navigation for the collection of oceanographic parameters. The design was divided into two fundamental parts: the first one has the objective of doing the loading and preprocessing of the correlation results; for this section the algorithms is implemented in the programmable system part. In the second part, the architecture to implement the correlation algorithms is designed and its implementation is in the FPGA programmable logic because it has the highest computational cost. To make the tests, real images obtained from databases were used and compared with results from a PC achieving a higher computational speed with a confidence of 60 % using 320×240 images. This first advance can be used for the creation of a more complete navigation system.

Índice general

1. Introducción **1**

1.1. Antecedentes 2

1.2. Descripción del problema 4

1.3. Objetivos 5

1.3.1. Objetivo general 5

1.3.2. Objetivos específicos 5

1.4. Justificación 5

1.5. Alcance y limitaciones 5

2. Marco teórico **6**

2.1. Visión Estéreo 6

2.1.1. Geometría epipolar 7

2.1.2. Búsqueda de la correspondencia 8

2.2. Formato de imagen *Bitmap* 11

2.3. Espacios de color 14

2.3.1. Escala de grises 15

2.3.2. *Red*, *Green* y *Blue* RGB() 15

2.3.3. *Hue*, *Saturation* y *Value* (HSV) 16

2.4. Algoritmos de preprocesado 17

2.4.1. Operador de punto 17

2.4.2. Operador local 18

2.5. Algoritmo de ordenación *Quicksort* 20

2.6. *Field Programmable Gate Array* (FPGA) 22

2.6.1. *Look-Up Table* (LUT) 23

2.6.2. Biestables (*flips-flops*) . . . 23
2.6.3. *Block Random Access Memory* (BRAM) . . . 25
2.6.4. *Application Processing Unit* (APU) . . . 25

3. Desarrollo de actividades **27**
3.1. Introducción al problema . . . 27
3.2. Obtención de los componentes . . . 28
3.3. Selección del *software* . . . 28
3.4. Diseño de los algoritmos . . . 30
3.4.1. Obtención de la información . . . 30
3.5. Diseño del hardware . . . 31
3.5.1. Problemas encontrados durante el diseño . . . 31
3.5.2. Diseño de los componentes . . . 33
3.6. Diseño del microcontrolador . . . 52
3.6.1. Archivo de cabecera *imeg.h* . . . 53
3.6.2. Archivo de código fuente *imeg.c* . . . 57
3.6.3. Función *main()* . . . 64

4. Resultados **67**

5. Conclusiones **72**
5.1. Trabajo a futuro . . . 72
5.2. Repositorio . . . 73

A. Código fuente para la creación de las cabeceras de archivo en C++ **75**
A.1. Cabeceras de archivo de imagen *.bmp* . . . 75

B. Código fuente del algoritmo de búsqueda de la correlación en C++ **77**
B.1. match() . . . 77

C. Código fuente para definir el archivo Make de la IP **80**
C.1. *Makefile* . . . 80

Bibliografía **82**

Índice de figuras

1.1. Resultados anteriores. 3

2.1. Geometría epipolar. 8
2.2. Búsqueda de correspondencia. 9
2.3. Búsqueda de correspondencia estéreo.. 10
2.4. Estructura de un archivo de imagen *.bmp*. 13
2.5. Representación de los espacios de color. 14
2.6. Operador local. 18
2.7. Ejemplos de máscaras convolutivas. 20
2.8. Estructura conceptual de un FPGA. 22
2.9. Estructura conceptual de una LUT 23
2.10. Configuración de las BRAM. 26

3.1. SoC Xilinx Zynq-7000 empleado para este proyecto. 29
3.2. Diagrama general. 30
3.3. Registros de 32 bits 34
3.4. Componente *match_control.vhd* 37
3.5. Proceso de creación de las ventanas iniciales 41
3.6. Secuencia de estados del componente *addr_generator* 43
3.7. Componente *get_addr.vhd* 45
3.8. Estructura del procesador. 46
3.9. Escritura de los registros 46
3.10. Componente *sad.vhd* 49
3.11. Componente *comparator.vhd* 51

3.12. Activación del puerto SD desde el FPGA. 58
3.13. Diagrama de flujo función *main()*. 65

4.1. Diagrama de bloques del proyecto. 67
4.2. Consumo de recursos. 68
4.3. Consumo energético. 68
4.4. Resultados de la implementación a imágenes de la base de datos. 70
4.5. Resultados de la implementación a imágenes subacuáticas. 71

5.1. Propuesta para el intercambio de información. 73

Índice de tablas

2.1. Cabecera de archivo de imagen. 12
2.2. Cabecera de la imagen *.bmp* 13
2.3. Tabla de verdad del *flip-flop S-R* 24
2.4. Tabla de verdad del *flip-flop S-R* 24
2.5. Tabla de verdad del *flip-flop D* 25

3.1. Especificaciones clave del FPGA. 28
3.2. Estados del componente *processor_buffer*. 47
3.3. Macros como funciones. 54
3.4. Macros para los atributos de la imagen. 55
3.5. Macros para definir las direcciones de las imágenes 55
3.6. Macros para las direcciones de los registros. 56
3.7. Macros para cada uno de los estados de procesamiento 56

4.1. Tiempos de procesamiento. 69

Capítulo 1

Introducción

La visión estéreo es el proceso de tomar dos imágenes capturadas de una escena, y hacer una estimación del modelo 3D de la escena encontrando los píxeles de correspondencia entre ambas imágenes. En la captura de la información se emplean un par de cámaras separadas mediante un desplazamiento horizontalmente con el objetivo de que en ambas imágenes haya una diferencia en la posición de los objetos de la escena capturada. A esta diferencia se le conoce como disparidad y el principal objetivo de estos algoritmos es encontrar la diferencia de esos valores.

En un proyecto anterior [1] se desarrolló una propuesta implementando la librería de OpenCV en una computadora personal y empleando el lenguaje de programación Python. Sin embargo, se busca que el sistema esté implementado en hardware tomando en cuenta las limitaciones de este, además de que sea capaz de emplearse en un sistema autónomo con independencia energética.

La propuesta de este proyecto es diseñar un sistema de visión que sea capaz de hacer la búsqueda de correspondencia de un par de imágenes, en una matriz de puertas lógicas programable en campo (FPGA por sus siglas en inglés). Los dispositivos FPGA tienen la potencia de computo y el bajo consumo de energía que permiten implementar sistemas autónomos en ambientes extremos, como es el medio submarino. En este trabajo se describirá la manera de implementar los algoritmos que normalmente requieren sistemas de computo tradicionales, como una PC debido a las necesidades de usar librerías diseñadas para lenguajes de alto nivel. Para lograr este propósito se emplearon imágenes de un tamaño de 320×240 píxeles a color. En el estado del arte se encontró que hay múltiples algoritmos que pueden ser útiles, pero que

tienen diferentes complejidades y algunos funcionan mejor en situaciones muy específicas, por lo que se necesita hacer la selección de los que puedan ser útiles.

1.1. Antecedentes

En los años recientes, el procesamiento de imágenes a despertado el interés de los investigadores de las ciencias computacionales por el gran número de soluciones en la que se aplica [2] [3] [4] [5], entre algunas de ellas están la medicina, diseño digital, investigación forense, conducción autónoma o entretenimiento.

Sin embargo, el procesamiento de imágenes tiene un alto costo computacional, esto puede deberse a que la imagen contiene mucha información, o también por los atributos que la definen, como su tamaño o el espacio de color. Tomando en cuenta estos factores que pueden afectar el rendimiento, se estima que hay que desarrollar una metodología especifica para un entorno en el que se va a trabajar y los parámetros de la imagen que se van a usar, ya que no hay forma de desarrollar un sistema que sea funcional en su totalidad para cualquier entorno y también puede contener información innecesaria que tan solo logra dificultar el procesamiento.

En algunas metodologías, por ejemplo cuando tan solo se busca reconocer formas en la imagen, es suficiente con usar imágenes en escala de grises reduciendo el espacio de color, y por lo tanto, el espacio de búsqueda. Se han desarrollado un sinfín de algoritmos y metodologías usando diferentes técnicas de programación e inteligencia artificial para mejorar los resultados y el rendimiento de los algoritmos [6] [7] [8] [9] [10], pero con todo esto no se ha podido desarrollar un sistema que sea totalmente funcional.

En [1] se implementa el desarrollo de un sistema que implementaba procesamiento de imágenes empleando la librería de OpenCV, la cual provee funciones para filtrar y extraer características de las imágenes, facilitando el tratamiento. Sin embargo, el sistema se desarrolló pensando que sea dependiente de una computadora, ya que tiene una alta capacidad de procesamiento con respecto a otros sistemas autónomos. Los resultados obtenidos se pueden apreciar en la **Figura 1.1** donde se puede observar el mapa de disparidad obtenido tras el procesamien-

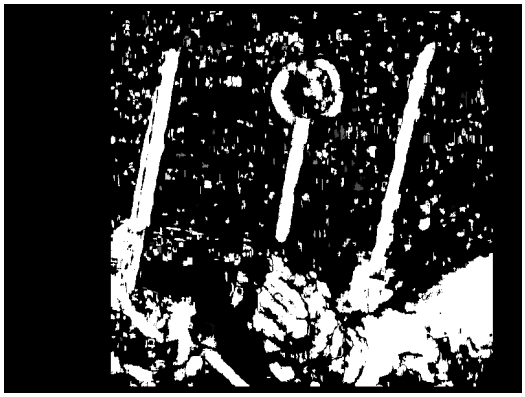


Figura 1.1: Mapas de disparidad obtenidos en un proyecto anterior.

to, donde se ve que hay demasiadas partículas y no están bien definidos los niveles de disparidad.

Ha habido diferentes aplicaciones de la visión estéreo en el ámbito de la investigación de entornos subacuáticos, sea para servir de asistencia en la navegación autónoma o en la recolección de datos. En [11] y [12] buscan obtener la batimetría desde la superficie del agua empleando vehículos aéreo no supervisados que recorren la superficie y tomando capturas para hacer una recreación en 3D de la superficie del mar y las costas. En estos trabajos se tiene que tomar en cuenta también que es necesario hacer un filtrado de las imágenes para hacer un ajuste del contraste y limitar las partículas que puedan hacer de obstáculo. Si bien este tipo de trabajos no es de lo que se anda buscando, es importante tomar en cuenta que la visión estéreo tiene también este tipo de aplicaciones así como que es necesario hacer un filtrado de las imágenes ya que al estar trabajando con entornos marinos y debido a la refracción de la luz y a la atenuación de la luz por medio del agua provoca que las imágenes obtenidas de estos entornos tengan problemas que puedan provocar falsos *matches* en la búsqueda de la correlación.

En [13] describe un vehículo autónomo llamado MARIS (*Marine Autonomous Robotics for InterventionS*), un vehículo propuesto para la asistencia en la investigación del fondo marino o para el rescate. Desde que el vehículo se tiene que desplazar por el fondo marino, es necesario que tenga un sistema de visión para el reconocimiento debajo del mar y la estimación de la profundidad. En este mismo trabajo se describe la implementación en hardware del sistema de visión estéreo así como también su debida calibración de las cámaras obteniendo como resultados 15 cuadros por segundo.

En [14] describe un vehículo que pueda servir de asistencia en la navegación e investigación submarina implementando un sistema de visión estéreo y reconocimiento para su recorrido por el fondo marino, empleando un Intel NUC colocado dentro de un bote que puede ser sumergido. En este trabajo se aborda el problema de que el agua y el material que envuelven las cámaras pueden provocar distorsión, además de que al navegar muy profundo las presiones aumentan, así que propone un algoritmo de calibración basado en el modelo *pinax* y un Análisis de Elemento Finito (*Finite Element Method*, FEM) para diseñar los compartimentos donde van a ser colocadas las cámaras que puedan soportar tal presión, así como también la profundidad máxima a la que se pueda navegar seguro.

En [15] propone un sistema de visión estéreo aplicado tanto en software como en hardware empleando un FPGA Altera Nios II. En este trabajo se toma en cuenta el hecho de que las imágenes capturadas debajo del mar tienen problemas de bajo contraste debido a que la luz se atenúa por el agua, así que se implementa un ajuste de los colores de las imágenes empleando la ley de Lambert-Beer. Hay un trabajo en [16] en el que específicamente busca hacer el ajuste de los colores en las imágenes capturadas en el fondo marino para hacer el resalte de los objetos.

1.2. Descripción del problema

La obtención de parámetros oceanográficos y recolección de datos del lecho marino supone la necesidad de inmersión de un sistema apropiado para realizar esa tarea. Para minimizar riesgos el sistema requiere una autonomía y en su caso, una navegación subacuática. La visión estéreo es un elemento que permite obtener parámetros oceanográficos o bien, funcionar como un sistema auxiliar para hacer la navegación.

En [1] funcionaba en un equipo de computo, usando la librería de procesamiento de imágenes OpenCV, con el lenguaje de programación Python, obteniendo resultados medios de acuerdo a lo que se buscaba. Sin embargo, el objetivo principal de este trabajo es desarrollar un sistema de visión estéreo que sea independiente al uso de librerías y sistemas de computo, ya que se busca que funcione en un dispositivo limitado en hardware como es el FPGA.

1.3. Objetivos

1.3.1. Objetivo general

Demostrar que algoritmos de alta complejidad computacional necesarias para la visión estéreo es posible implementarlo en sistema autónomo, que contribuya en la obtención de parámetros oceánicos.

1.3.2. Objetivos específicos

- Analizar la complejidad de los algoritmos para determinar el proceso que requiera mayor poder de cómputo.
- Desarrollar e implementar los algoritmos de procesado con más carga computacional en el FPGA.
- Desarrollar e implementar la interacción de sistemas que permitan el procesado de las imágenes.

1.4. Justificación

El uso de otra estrategia de obtención de datos del entorno haría que la complejidad computacional, el coste monetario y el coste energético aumente. El objetivo es crear un vehículo que sea usado para la recolección de parámetros subacuáticos, el implementar un FPGA otorgaría la suficiente autonomía y un gran poder de cómputo para que este vea reducido su tamaño y que disponga de autonomía suficiente para que la intervención humana sea la mínima, sobre todo en entornos peligrosos.

1.5. Alcance y limitaciones

- El sistema de cómputo es de gama baja.
- La disponibilidad de hardware es limitado para asegurar la autonomía del sistema.
- Las imágenes empleadas son de baja resolución.

Capítulo 2

Marco teórico

2.1. Visión Estéreo

La visión estéreo consiste en el proceso de tomar dos imágenes y estimar un modelo 3D de la escena encontrando los píxeles de correspondencia en la imagen, y convertir sus posiciones 2D en una profundidad 3D. La metodología está basada en la manera que los humanos tienen posicionados los ojos y como son controlados, nuestros cerebros usualmente reciben imágenes similares de una escena tomadas en un punto concreto con un desplazamiento horizontal [17]. Con este desplazamiento, se crea una diferencia entre ambas imágenes y es posible hacer una estimación de la profundidad. El objetivo principal de estos algoritmos es encontrar esta diferencia haciendo comparaciones entre los píxeles entre ambas imágenes para buscar la correspondencia, también conocida como *disparidad*. Estos algoritmos se pueden dividir principalmente en dos tipos [7] [18]: los *algoritmos locales* y los *algoritmos globales*.

Los algoritmos locales emplean una ventana de búsqueda que hace la correlación entre ambas imágenes, obteniendo una "distancia" por medio de una función de coste. Entre las funciones empleadas están: *Suma de Diferencias Absolutas* (SAD), *Suma de Diferencias al Cuadrado* (SSD) y la *Transformada de Census* [18]. Este tipo de algoritmos son los que más se suelen recomendar y emplear para su uso en un sistema en tiempo real, pero acarrea una serie de problemas tales como:

- **Ruido.** Puede verse como seria distintos niveles de iluminación en la escena, o también

un mal ajuste de las cámaras podrían hacer que las imágenes obtenidas se vean afectadas.

- **Regiones sin texturas.** También es conocido como *problema de apertura* [19]. Esto hace que obtener información de la región se complique por no haber elementos del cual basarse.
- **Profundidad discontinua.** El suavizado espacial de la escena, puede provocar que los objetos no se diferencien entre si, provocando que dé problemas al momento de obtener los mapas de profundidad.
- **Oclusión.** Los píxeles ocluidos en una de las vistas no deberían de emparejarse con los píxeles de la otra imagen, generando partículas en las imágenes resultantes que comúnmente se conocen como *false targets* [17] o *false matches* [20] [21].

Esto problemas podrían provocar que los mapas de disparidad contengan partículas (como se puede observar en los resultados anteriores en **Figura 1.1** del primer capitulo). Algunos de estos problemas pueden ser corregidos rectificando las imágenes antes, o aplicando un filtro a la imagen resultante como puede ser el de mediana para reducir el número de partículas en la imagen [22] [23] [24]. Además, por la misma forma de las cámaras pueden agregar un poco de distorsión a las imágenes y, ademas se requiere encontrar la relación entre los puntos 3D de un entorno real y los puntos de las imágenes [25]. Estos problemas anteriormente mencionados, se pueden resolver mediante un calibración de las cámaras, con la intención de encontrar las matrices de corrección y de características con las cuales ajustar las imágenes de entrada.

Por el otro lado, los algoritmos globales trabajan con la imagen entera, evitando la mayoría de problemas anteriormente mencionados, a coste de incrementar mucho el coste computacional [22].

2.1.1. Geometría epipolar

Es la correspondencia de un punto de una imagen con el punto de la otra imagen. En el caso de la visión estéreo, se usa otra información adicional para definir la geometría, como la posición y la calibración de las cámaras. En la **Figura 2.1** se puede ver como es la triangulación de las cámaras, y como es que se define la geometría epipolar; las *lineas epipolares* se definen

como la línea que van desde el centro óptico O_i de la cámara original, hasta un punto infinito P_∞, esta linea en complemento con la segunda linea epipolar que va desde el centro óptico de la segunda cámara O_d, forma un plano conocido como *plano epipolar*.

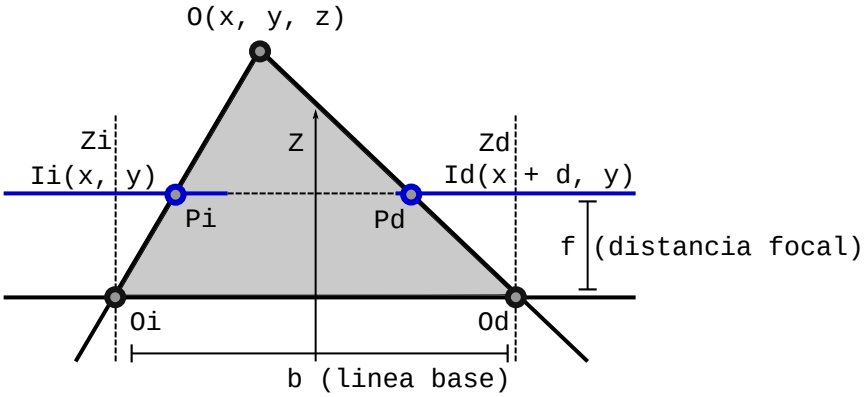


Figura 2.1: Representación de la triangulación de las cámaras que definen la geometría epipolar.

2.1.2. Búsqueda de la correspondencia

Un algoritmo local funciona mediante una ventana, el cual va buscando los píxeles de correspondencia entre las imágenes izquierda y derecha mediante una función de coste. En la posición de la imagen $I(x, y)$ en que sus píxeles vecinos son muy similares, entonces la función de coste retorna un valor mínimo, Este proceso es hecho por cada nivel de disparidad. En la **Figura 2.2** se muestra el proceso de búsqueda.

Entre más niveles de disparidad la estimación de la profundidad es más precisa, pero esto también aumenta el tiempo de procesamiento. En la ventana se aplica una función de coste, que obtiene un valor local y temporal que se almacena en un *buffer*. La función más ampliamente usada para estimar los costes es *Sum of Absolute Differences* la cual queda representada de la siguiente manera (**Ecuación 2.1**) [20]:

$$SAD(x, y, d) = \sum_{x,y \in W} |L(x, y) - R(x - d, y)| \tag{2.1}$$

donde $SAD(x, y, d)$ corresponde al *buffer* que se almacena los valores locales obtenidos por la

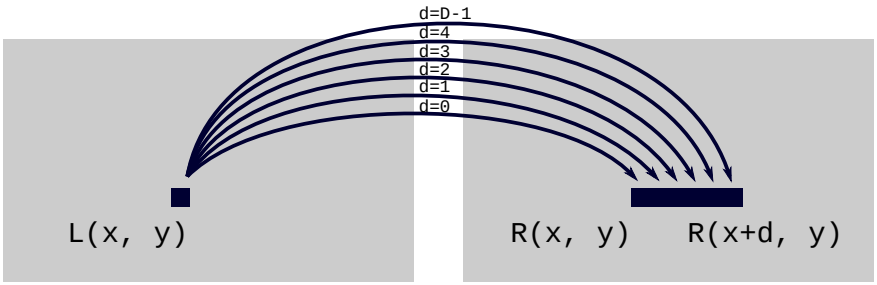


Figura 2.2: Proceso por el cual se hace la búsqueda de correspondencia entre ambas imágenes. $L(x, y)$ representa la imagen izquierda, mientras que $R(x, y)$ representa la imagen derecha. Por cada ventana de la imagen izquierda se recorren $D - 1$ niveles en la imagen derecha buscando la correlación entre ambas.

función aplicada a ambas imágenes, $L(x, y)$ corresponde a la ventana de la imagen izquierda y $R(x - d, y)$ a la ventana de la imagen derecha. Esta función es ampliamente usada debido a que solo involucra una operación de suma, resta y valor absoluto, por lo que es muy sencilla de implementar.

Otra función que se ha visto su implementación es *Sum of Squared Differences* (SSD) (**Ecuación 2.2**) [20]:

$$SSD(x, y, d) = \sum_{x,y \in W} (L(x, y) - R(x - d, y))^2 \tag{2.2}$$

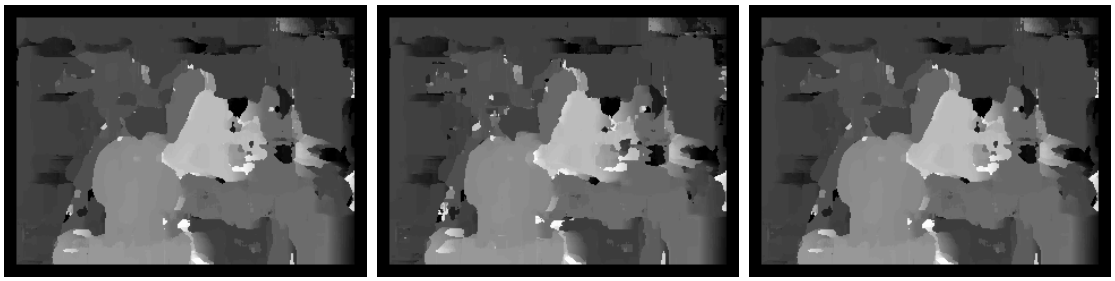
La ecuación es similar a la **Ecuación 2.1**, la diferencia radica en que se usa una operación de potencia en vez de un operador de valor absoluto, y es esta operación que provoca que el coste computacional aumenta mucho [20]. Por último, otra función de estimación de coste que se suele emplear es *Normalized Cross-Correlation* representado en la siguiente expresión (**Ecuación 2.3**) [20] [26]:

$$NCC(x, y, d) = \frac{\sum_{x,y \in W} |L(x, y) - R(x, y - d)|}{\sqrt{\sum_{x,y \in W} L^2(x, y) \sum_{x,y \in W} R^2(x, y - d)}} \tag{2.3}$$

Esta última función está basada en el coeficiente de correlación empleado en la estadística para buscar la relación entre dos variables. Entre más relación tengan estos datos con respecto a su eje, estos van a tender a -1 o 1, mientras que más se acercan a 0, entonces tienen correlación débil. En este caso se busca la correlación que hay entre los dos píxeles de ambas imágenes.

De entre los elementos de el arreglo generado, se obtiene el mínimo argumento $d(x, y) = argmin\ SAD(x, y, d)$, el cual consiste en buscar la posición que contiene el valor mínimo. La posición en el arreglo, que se entiende como un número entero que va de 0 hasta $D - 1$, siendo D el número de niveles de disparidad, se convierte en el nuevo nivel de gris de la imagen de salida.

En el **Anexo B** se muestra un ejemplo escrito en C++ implementando el algoritmo de búsqueda de correlación empleando la función *SAD*, en el que simplemente consiste en múltiples ciclos *for* que hacen los recorridos por la imagen, las ventanas y cada uno de los niveles de disparidad, lo que estaría variando es la función de estimación de coste que cambiaría de acuerdo al caso. La complejidad de este algoritmo es alta y no está pensada para que sea implementada en un sistema. Ejemplos de la implementación del mismo se puede observar en la **Figura 2.3** empleando imágenes ya rectificadas obtenidas de la base de datos de *Middlebury* rescatado del sitio `https://vision.middlebury.edu/stereo/data/` [23] [27].



(a) *SAD* (b) *SSD* (c) *NCC*

Figura 2.3: Resultados de aplicación de las funciones de estimación de coste empleando el código mostrado en el **Anexo B**. Los niveles máximos de disparidad usado son $d = 64$ y el tamaño de la ventana $w = 11 \times 11$.

2.2. Formato de imagen *Bitmap*

Las imágenes que se suelen usar tienen colocados los píxeles dispuestos en forma de matriz; a este tipo de imágenes se les conoce como *mapa de bits*. Estas suelen estar almacenadas en diferentes formatos, en el que cada uno tienen diferentes algoritmos de compresión y ordenación de los píxeles que están contenidos. El sistema operativo hace el reconocimiento de cada uno de los formatos mediante la extensión que aparece al final del nombre del archivo, indicándole con que aplicación se tiene que abrir. Esta aplicación hace la lectura de la información y la despliega siguiendo o tomando en cuenta el algoritmo de compresión que tiene cada uno de los formatos. La compresión de las imágenes ha sido necesario desde que se necesita transitar esta información a través de la internet, y entre menos peso tenga, hay un ahorro mayor de datos de descarga [28], sin embargo, en los últimos años se ha visto incrementado el uso de imágenes vectoriales en las que no se almacenan todo el arreglo de píxeles, sino se almacena un conjunto de coordenadas que definen puntos y lineas que, al ser desplegada por una aplicación, hace el dibujado siguiendo estas instrucciones. Son complejas de procesar, pero ha habido una disminución significativa en el peso de la imagen, además de que tienen la ventaja de que no pierden calidad al aplicar una transformación como puede ser el escalado o la rotación.

En el área de procesamiento de imágenes, solo es necesario el arreglo de píxeles para el procesamiento y si se toma uno de estos formatos que tiene la información comprimida, solo complica el procesado. Hay librerías que ayudan en el manejo de imágenes, pero cuando se requiere trabajar con un sistema muy especifico, se tiene que optar por otra opción.

El formato *.bmp* es un formato de imagen creado por Microsoft para el sistema operativo Windows. El nombre viene del inglés *bitmap*, y no es más que una matriz en la que sus dimensiones son la anchura y altura de la imagen, y en cada celda se almacena el píxel correspondiente [28]. Por lo general no están comprimidas, así que las imágenes en este formato son muy pesadas ya que se almacenan todos los píxeles, incluyendo aquellos que se repiten.

Para que la lectura de la imagen por la aplicación sea posible, el formato plantea una estructura en la que se deposita la información referente a la imagen. Esta información está

almacenada en las cabeceras (o en ingles *header*) y contiene los atributos que definen la imagen como sus dimensiones, profundidad de bits y paletas. De manera básica, los archivos con extensión *.bmp* se pueden dividir en tres partes: cabecera de archivo, cabecera de imagen y datos, con un peso total de 54 bytes para las cabeceras y el resto es para los píxeles.

La primera cabecera almacena información relacionada con el archivo y es empleada por el sistema operativo para indicar el modo en el que se va a procesar la información que esta contiene. Cada uno de los atributos se definen en la **Tabla 2.1**. Algunos de los valores vistos en la tabla cambian conforme se define la imagen, los valores vistos en la tabla son por defecto.

Nombre	Tamaño	Valor	Descripción
FILE_TYPE	*16 bits*	0x4D42	Tipo de archivo, normalmente tiene el valor BM indicando que es BMP.
FILE_SIZE	*32 bits*	0xXXXXXXXX	Tamaño total del archivo en bytes.
RESERVED	*32 bits*	0x00000000	Reservado, normalmente 0.
OFFSET_DATA	*32 bits*	0x00000036	Dirección donde se encuentra localizado el primer píxel.

Tabla 2.1: Cabecera que contiene información relacionada con el archivo de imagen. Las equis representan cualquier valor.

La siguiente cabecera permite definir los atributos de la imagen. Esta contiene información relacionada con la profundidad de bits, la resolución y las dimensiones de esta. En la **Tabla 2.2** se definen cada uno de los registro de esta cabecera. Como en el caso anterior, los valores también están por defecto y estos cambian de acuerdo a la imagen.

Para ejemplificar esto, en la **Figura 2.4** se muestran los primeros 112 bytes que conforman un archivo de imagen de 24 bits, de los cuales, los primeros 54 bytes corresponden a la información de las cabeceras, mientras a partir de la dirección 36_h comienza la información de los píxeles. Cabe recalcar que en las imágenes de 24 bits, cada 3 bytes conforman un píxel comenzando por el rojo, luego el verde y al final el azul. La resolución horizontal tiene que ser un

Nombre	Tamaño	Valor	Descripción
HEADER_SIZE	*32 bits*	0x00000028	Tamaño de esta cabecera, normalmente de 40 bytes.
IMG_WIDTH	*32 bits*	0xXXXXXXXX	Altura de la imagen.
IMG_HEIGHT	*32 bits*	0xXXXXXXXX	Anchura de la imagen.
IMG_PLANES	*16 bits*	0x0001	Número de planos de la imagen.
BITS_PER_PIXEL	*16 bits*	0xXXXX	Profundidad de bits.
COMPRESSION	*32 bits*	0x00000000	Método de compresión de la imagen.
SIZE_IMAGE	*32 bits*	0xXXXXXXXX	Tamaño total del arreglo de píxeles.
X_RESOLUTION	*32 bits*	0x00000EC4	Resolución horizontal.
Y_RESOLUTION	*32 bits*	0x00000EC4	Resolución vertical
COLORS_USED	*32 bits*	0xXXXXXXXX	Número de colores por paleta.
COLORS_IMPT	*32 bits*	0xXXXXXXXX	Número de colores importantes por paleta.

Tabla 2.2: Cabecera que contiene los registros para la definición de los atributos de una imagen. Las equis representan cualquier valor.

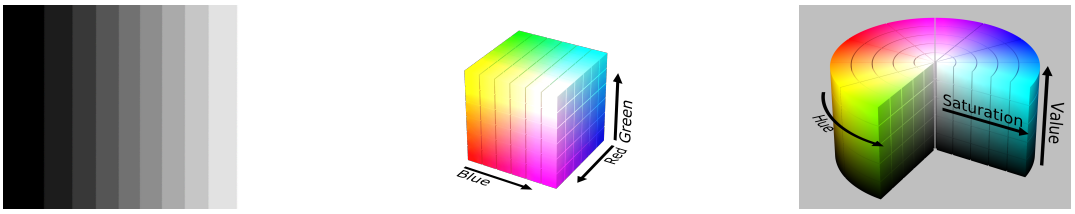
múltiplo de 4, en caso de que no se cumpla esta condición, entonces se rellena los bytes faltantes con píxeles en 0.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	42	4D	3E	45	00	00	00	00	00	00	36	00	00	00	28	00
1	00	00	3E	00	00	00	5E	00	00	00	01	00	18	00	00	00
2	00	00	00	00	00	00	8E	0E	00	00	8E	0E	00	00	00	00
3	00	00	00	00	00	00	3C	54	60	3C	54	60	49	56	60	00
4	00	00	00	3C	5C	08	52	7A	00	3C	5C	06	41	70	06	41
5	70	00	3C	5C	00	3C	5C	00	3C	5C	00	2C	4C	00	2C	4C
6	00	3C	5C	00	3C	5C	00	2C	4C	60	80	94	3C	54	60	49

Figura 2.4: Estructura de un archivo de imagen formato *.bmp*. A partir de la dirección 54_d (36_h) comienza la información de los píxeles.

2.3. Espacios de color

De manera común, las pantalla a color de diferentes dispositivos muestran los colores mediante la combinación de tres principales: rojo, verde y azul [29] [28]. A esta estructura de almacenar los colores de esta manera se le conoce como *espacio de color*, y no es más que representar los colores mediante canales en el que cada uno tiene un nivel diferente de intensidad que, en conjunto, permite la creación de los demás colores. Esto queda mejor ilustrado en la **Figura 2.5**. Sin embargo, al estar limitados por esta configuración no ha sido posible hacer la representación de todos los colores en su espectro electromagnético mediante un medio como una pantalla, además de que la percepción de color puede variar de persona a persona y entre las diferentes especias animales. El espacio de color más común es el RGB (de *red*, *green* y *blue*) [28], ya que permite la representación de una imagen mediante tres canales y facilita su desplegado en un pantalla. Sin embargo, en el ámbito de procesamiento de imágenes, en ocasiones va a ser necesario el manejo de otros espacios además de este.



(a) escala de grises (b) rgb (c) hsv

Figura 2.5: Representación de los espacios de color. Las tres imágenes son cortesía de *Creative Commons*.

Cuando se almacena un color en una estructura, se hace más eficiente que sea almacenado como una valor entero que va desde 0 a 255 tratándose de imágenes de 8 bits. Las imágenes que normalmente rondan la internet suelen ser de 24 bits, es decir, tienen tres canales de 8 bits en el que cada uno se almacenan los niveles para cada color RGB, y en casos más específicos, hay de 32 bits que permiten un canal extra para especificar la transparencia.

2.3.1. Escala de grises

También conocido normalmente como (de manera errónea) *blanco y negro*, es una manera de representar los colores de la imagen mediante niveles de gris. A diferencia de las imágenes RGB, estas solo almacenan un canal de color, sin embargo, es muy útil cuando solo se requiere información como los cambios de niveles de gris usado para el reconocimiento o resalte de regiones en la imagen, mejorando así la velocidad de procesamiento. La manera en la que se puede convertir una imagen RGB de 24 bits a una de escala de grises de 8 bits está representado en la **Ecuación 2.4**:

$$G(x, y) = [F(x, y, B) \times 2 + F(x, y, G) \times 4 + F(x, y, R)]/7 \tag{2.4}$$

donde *G(x, y)* representa el píxel nuevo, y *F(x, y)* el píxel de la imagen que se está convirtiendo. Los valores B, G, y R representan los canales azul, verde y rojo respectivamente.

Las imágenes en escala de grises suelen ser de 8 bits, almacenando valores con rango de 0 (negro) hasta el 255 (blanco), y los niveles intermedios representando grises. Son un total de 256 valores diferentes los que se puede representar mediante esta estructura.

2.3.2. *Red*, *Green* y *Blue* RGB()

Es por mucho el espacio de color más usado ya que está soportado por muchos dispositivos hasta el día de hoy [28]. Esta estructura guarda los colores en tres canales en el cada uno maneja 256 niveles de color, y cada uno representa un nivel de rojo, verde y azul respectivamente. La combinación de estos permite la representación de los demás colores. Por ejemplo, para crear el color blanco se ponen los tres canales a su máximo valor y es representado como $[255, 255, 255]$ mientras que el negro es ausencia de color y se representa como $[0, 0, 0]$. Para representar otros colores como violeta puede ser $[163, 73, 164]$ mientras el color amarillo se hace con $[255, 255, 0]$. Si las imágenes son de 24 bits (8 bits por canal), entonces el total de colores es de $16,777,216$, sin embargo, esto no representa el total de colores reales.

2.3.3. *Hue*, *Saturation* y *Value* (HSV)

Es otra manera en la que se pueden estructurar los colores manejando tres canales también, sin embargo, estos representan otras unidades diferentes a la de RGB. Estos tres canales son: matiz (en inglés es *hue*) que representa el color, saturación (en ingles es *saturation* o en ocasiones *chroma*) representa la cantidad o la intensidad del color, y valor (en inglés *value*) representa la luminiscencia [28].

La representación se hace empleando diferentes unidades. En la **Figura 2.5** se puede observar que la representación se hace mediante un cilindro, en el que el angulo indica la matiz, mientras que por medio de valores de entre 0 y 1 se representa los diferentes niveles de saturación y luminiscencia.

Para convertir una imagen RGB a este espacio de color se puede hacer empleando las siguientes ecuaciones. Para estimar el *valor*, se toma el valor máximo de entre los canales de rojo, verde, y azul, y esto queda representado mediante la **Ecuación 2.5**:

$$V = max(R, G, B) \tag{2.5}$$

donde R, G, y B representa cada uno de los canales rojo, verde y azul respectivamente.

Para calcular la *saturación* se utiliza el *valor* calculado anteriormente (denotado V) en el que se requiere calcular el valor mínimo de entre los tres colores R, G y B. Representado en la **Ecuación 2.6**:

$$S = \begin{cases} \frac{(V - min(R,G,B)}{V} & si\, V \neq 0 \\ 0 & si\, V = 0 \end{cases} \tag{2.6}$$

Finalmente, para calcula la *matiz* se requiere el *valor* (denotado V), pero además, es necesario hacer la suma de el angulo para ajustar la *matiz* a su respectivo rango, y esto va a depender de que canal de entre los tres es el mayor. Los colores van de 0 a 360 grados, en el que de 0 a 119

representa el rojo, de 120 a 239 representa el verde, y el resto representa el azul. La **Ecuación 2.7** permite hacer esta conversión:

$$H = \begin{cases} \frac{60*(G-B)}{V-min(R,G,B)} & si\ V = R \\ \frac{120+60*(B-R)}{V-min(R,G,B)} & si\ V = G \\ \frac{240+60*(R-G)}{V-min(R,G,B)} & si\ V = B \\ 0 & si\ R = G = B \end{cases} \tag{2.7}$$

Si este espacio de color es almacenado en una imagen empleando tres canales de color de 8 bits cada uno, la *matiz* debe de ser dividida entre 2 para que pertenezca al rango de 0 a 255 niveles, perdiéndose calidad en el color. Los otros dos canales pueden ser almacenados directamente. Este espacio de color es muy útil cuando se quiere hacer una segmentación por medio del color, ya que solo teniendo el color deseado en un solo canal es mucho más sencillo que manejar a diferencia del RGB en el que se tiene que considerar los tres.

2.4. Algoritmos de preprocesado

Las imágenes de entrada suelen tener ruido producto de la configuración de las cámaras o debido a la calidad del entorno donde se hizo la captura, además de que puede contener información que no sea necesaria. En la parte de preprocesado, consiste en eliminar o disminuir la información que solo es un obstáculo para el procesado, además de que ayuda a reducir los tiempos de procesado. La reducción de ruido puede consistir en aplicar un filtro de difuminado para reducir el número de partículas conllevando también una reducción de precisión, mientras que la reducción de características conlleva en eliminar información innecesaria o reducir el número de canales en la imagen. Estos algoritmos se pueden dividirse en dos tipos dependiendo del operador: operador de punto y operador local.

2.4.1. Operador de punto

Los operadores de punto solo operan el píxel ubicado en la coordenada $I(x, y, c)$ de la imagen, siendo x la columna, y la fila y c el canal de color. Para aplicar estos operadores solo es suficiente

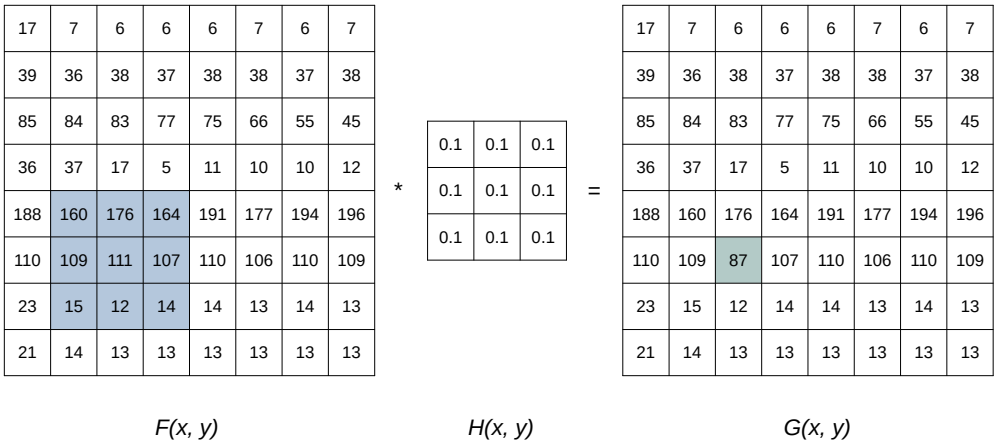


Figura 2.6: Operador local, en el que se ilustra que en la imagen de entrada se están seleccionando los píxeles en *F(x, y)* (marcado con color azul), y a esto se aplica una máscara $H(k, l)$ empleando la 2.8 que da como resultado una nueva imagen indicada con *H(x, y).*

con recorrer el arreglo que contiene los píxeles e ir aplicando el operador a cada uno de los elementos. Entre algunas funciones básicas están la conversión a escala de grises, inversión de colores, ajuste de brillo y contraste y la segmentación [29].

2.4.2. Operador local

En el caso de los operadores locales, se toman los píxeles que se ubican en los alrededores del píxel $I(x, y)$ para estimar el píxel resultante de la nueva imagen [29] mediante una matriz empleado a modo de máscara o *kernel* tal y como se puede observar en la **Figura 2.6**. La función básica que se suele emplear en estos operadores es la convolución representada en la **Ecuación 2.8**.

$$G(x, y) = \sum_{k,l} F(x + k, y + l) \times H(k, l) \tag{2.8}$$

De la ecuación anterior se puede obtener el siguiente pseudocódigo:

```
CONV(image, width, height, W, K)
    aux = 0
    new_image[,] = 0
```

```
4
5       for x = W to width
6           for y = W to height
7               for k = -W to W
8                   for l = -W to W
9                       aux += image[x + i, y + j] * K[k, l]
10
11              new_image[x, y] = aux / size of K
12      return new_image
```

donde se pude observar que se están empleando cuatro ciclos *for* en total para hacer los recorridos: los dos primeros es para recorrer todos los píxeles de la imagen y los hace tanto por la anchura y la altura, mientras que los dos más internos hace los recorridos por la ventana. En el ciclo más interno hay una operación de multiplicación del píxel actual en *F(x, y)* y el elemento de la máscara en *H(i, j)*, el cual es acumulado. Finalmente, el resultado de acumulación se multiplica por un coeficiente antes de asignar a la imagen nueva en *G(x, y)*.

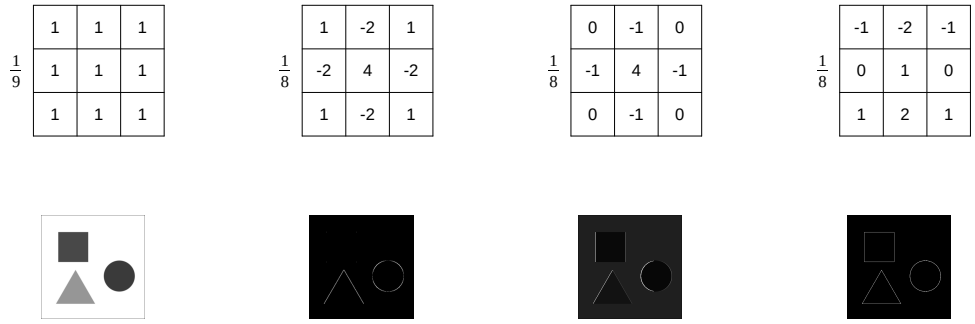
De este operador, salen dos nuevos conceptos: ventana y máscara (también conocido como *kernel* en inglés).

Ventana

Son los aquellos píxeles que se encuentran alrededor de un píxel ubicado en la coordenada *F(x, y)*. El tamaño de la ventana especifica la cantidad de píxeles que se está leyendo ignorando el píxel central. Por ejemplo, si son ventanas de 5×5, se están leyendo dos píxeles a los lados, arriba y abajo del píxel *I(x, y)*. En ocasiones el píxel no se toma como si estuviera centrado, en este caso solo cambia la dirección en la que se lee los píxeles y la cantidad de ellos.

Máscara o *kernel*

En los algoritmos convolutivos, la máscara representa una matriz que contiene los coeficientes de filtrado [29], dando como resultado una imagen en que la información fue alterada por la máscara y la función especificada en la **Ecuación 2.8**. El tipo de filtrado depende de la máscara



(a) *box* (b) *corner* (c) repujado (d) bordes

Figura 2.7: Ejemplos de máscaras o *kernel* usados para el filtrado de una imagen.

que se esté usando y la calidad de los resultados depende del tamaña de esta y de la ventana, sin embargo, ventanas demasiado grandes solo provoca que la información de las regiones de la imagen se distorsione. En la **Figura 2.7** se muestran varios ejemplos de máscaras que se usan para el filtrado o transformación de la imagen.

2.5. Algoritmo de ordenación *Quicksort*

Muchos problemas relacionados con la computación (o en general el día a día) requiere la ordenación de un conjunto de datos. Cuando se trata con listas pequeñas esto no representa un problema, sin embargo, cuando las listas son extensas y los datos a ordenar son muy largas, entonces crea una situación complicada que puede resolverse mediante un algoritmo de ordenación. Cada uno de estos algoritmos tienen una complejidad diferente y este puede hacerse más optimo en situaciones especificas. Normalmente los algoritmos en su peor caso tienen una complejidad de $\Theta(n^2)$ [30], esto es debido a que estos algoritmos tienen ciclos internos que se encargan de hacer los recorridos por el arreglo y hacer las comparaciones con cada uno de los elementos para hacer el reemplazo.

El algoritmo *Quicksort* presenta una complejidad algorítmica de $\Theta(n^2)$ en su peor caso, mientras que en el caso promedio y el mejor caso, la complejidad algorítmica es de $\Theta(n\ lg\ n)$ [30], esto lo hace uno de los mejores algoritmos para usar en la ordenación. Este algoritmo usa el paradigma de programación *divide y conquista*, el cual consiste en dividir el problema en dos partes: dividir el problema (*divide*) y resolver cada una de las partes (*conquista*). La división se

hace siguiendo una metodología recursiva; se divide el arreglo en dos partes, tomando el indice p como la base y el indice q como el máximo del subarreglo, el indice r indica el máximo del arreglo y el máximo que va a tomar el segundo subarreglo. Como es un algoritmo recursivo, el valor de la variable q va a depender de cuantos elementos dentro del subarreglo fueron ordenados, para que al siguiente nivel este se convierta en el pivote para la siguiente división. El pseudocódigo de esta partición se muestra a continuación:

```
1  QUICKSORT(A, p, r)
2  if p < r
3  q = PARTITION(A, p, r)
4  QUICKSORT(A, p, q - 1)
5  QUICKSORT(A, q + 1, r)
```

El siguiente pseudocódigo es la parte clave del algoritmo. Se vuelven a calcular los indices de los subarreglos, y ordena cada uno de estos. Se toma el subarreglo y se hace las comparaciones para encontrar el mínimo y se va incrementando el indice i cada vez que encuentra uno menor, una vez encontrado se hace el intercambio con el dato original que tiene el subarreglo. Al final se hace el intercambio en el subarreglo y se retornar el contador i que se va a convertir en q para hacer la partición en el siguiente nivel.

```
1  PARTITION(A, p, r)
2  x = A[r]
3  i = p - 1
4  for j = p to r - 1
5  if A[j] <= x
6  i = i + 1
7  swap A[i] with A[j]
8  swap A[i + 1] with A[r]
9  return i + 1
```

La eficiencia del algoritmo depende de si la partición está balanceada o no, o también si las particiones tienen la misma cantidad de elementos. Si la partición está desbalanceada la recurrencia es $T(n) = T(n-1) + T(0) + \Theta(n)$, esto quiere decir de que en las dos particiones iniciales: una se está llevando $n-1$ elementos mientras que el la otra partición se está llevando 0 elementos, indicando que el peor caso para *Quicksort* es cuando el arreglo ya está ordenado,

ya que una de las particiones está tomando todos los elementos mientras que la otra partición está vacía. Por el otro lado, el caso ideal es cuando la recurrencia es $T(n) = 2T(n/2) + \Theta(n)$ haciendo que las particiones estén balanceadas, y por lo tanto, se presente el mejor caso.

2.6. *Field Programmable Gate Array* (FPGA)

Una Matriz de Puertas Lógicas Programables en Campo (FPGA por las siglas en ingles *field programmable gate array*)[31] es un dispositivo que contiene un arreglo de dos dimensiones de interruptores (en inglés *switches*) que son programables mediante un lenguaje descriptor HDL especializado como pueden ser VHDL o Verilog. Cada una de la celdas lógicas pueden ser reconfiguradas para ejecutar una tarea simple, y cada uno de los interruptores se configura para establecer una conexión entre cada una de las celdas, como se puede observar conceptualmente en la **Figura 2.8**. Un diseño personalizado se puede hacer configurando cada una de las celdas, y selectivamente ir conectando cada uno de los interruptores para tener el funcionamiento deseado. Una vez que el diseño y la síntesis estén completas, solo se puede hacer la conexión por medio de un cable para ”descargar” esta configuración a el FPGA y obtener un circuito personalizado. Un FPGA pude llegar a tener hasta cientos de miles de estas celdas e interruptores que pueden dar sustento a muchas alternativas para hacer circuitos personalizados a una necesidad [32].

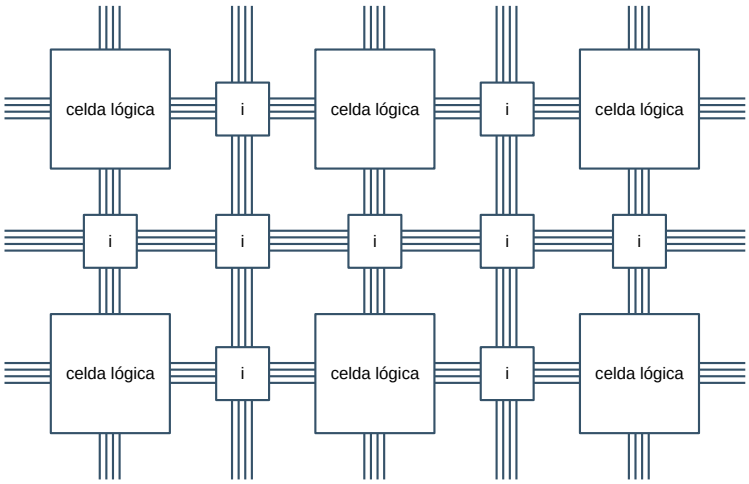
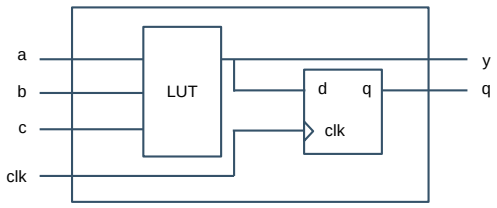


Figura 2.8: Estructura conceptual de un FPGA, en el que se observa las celdas lógica interconectadas mediante interruptores (en la figura representado por i).

2.6.1. *Look-Up Table* (LUT)

Cada una de las celdas lógicas del FPGA contiene lo que se le conocen como tablas de consulta y *flip-flops* [32] [31]. Estas tablas de consulta son las LUT's (del inglés *look-up table*) y no es más que una tabla de verdad almacenada en memoria, que hace la combinatoria de las entradas n produciendo una salida, respaldada por un *flip-flop* normalmente tipo D (DFF), tal y como está representado en la **Figura 2.9**.



(a) LUT de tres entradas.

a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(b) Tabla de verdad.

Figura 2.9: Estructura conceptual de una LUT de tres entradas y su correspondiente tabla de verdad que responde a la función $y = a + b \times c$.

En la terminología de Xilinx, un CLB (*Configurable Logic Blocks*) contiene un par de *slices* que a su vez contiene en total cuatro LUTs [33]. Cada una de las *slices* que contiene cada bloque están colocadas en disposición de columnas y cada una de las columnas tienen su propio acarreo, además de que cada *slice* no tiene una conexión directa entre si. Esta información fue revisada para la FPGA Zybo Zynq XC7Z010-1CLG400C.

2.6.2. Biestables (*flips-flops*)

Un *flip-flop* es un circuito que puede almacenar dos posibles estados estables (de ahí que en español se les conozca como ***biestables***) por un tiempo indefinido hasta que se reciba una señal de reloj que le indique que se tiene que cambiar el estado a uno siguiente [32]. Estos circuitos forman parte de la lógica secuencial y están sincronizados a las señales de reloj. Los hay de diferentes tipos, y la diferencia reside en que cada uno actúa de diferente manera a la señal de reloj que recibe, estos tipos son : *S-R*, *J-K* y *D*.

Biestable tipo *S-R*

Este biestable toma el nombre por *set-reset*, esto significa que, además de la señal de reloj, también toma dos señales más: una para establecer el valor (*set*) y la otra es para reiniciar el valor almacenado (*reset*). Estas señales son mutuamente excluyentes, indicando que no es posible recibir las señales *set-reset* al mismo tiempo, produciendo un estado invalido. se puede observar en la tabla de verdad este funcionamiento.

Entradas		Salidas		
S	R	Q	Q'	Comentario
0	0	Q	Q'	No hay cambio
0	1	0	1	RESET
1	0	1	0	SET
1	1	X	X	Inválido

Tabla 2.3: Tabla de verdad del *flip-flop S-R.*

Biestable tipo *J-K*

Similar a el tipo *R-S*, salvo que la diferencia es que cuando ambas señales *J-K* se establecen en alto, no se produce un estado invalido, si no más bien se intercambian ambos estados.

Entradas		Salidas		
J	K	Q	Q'	Comentario
0	0	Q	Q'	No hay cambio
0	1	0	1	RESET
1	0	1	0	SET
1	1	Q'	Q	Conmutar

Tabla 2.4: Tabla de verdad del *flip-flop J-K.*

Biestable tipo *D*

El más simple en comparación con los anteriores. Cuando señal de entrada *D* es alta, entonces se establece el estado a 1, en caso contrario el estado se reinicia a 0.

Entrada	Salidas		
D	Q	Q'	Comentario
0	0	1	RESET
1	1	0	SET

Tabla 2.5: Tabla de verdad del *flip-flop D*

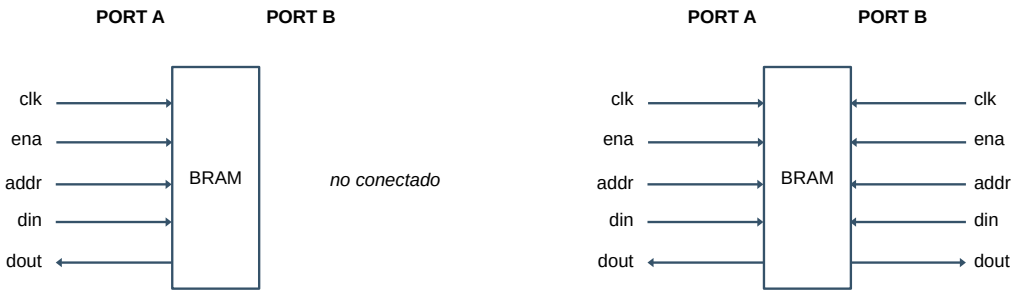
2.6.3. *Block Random Access Memory* (BRAM)

Son empleados por el FPGA para almacenar una gran cantidad de datos dentro y poder acceder directamente a ellos sin necesidad de hacer las lecturas a una unidad externa como puede ser la DDR. Son unidades rápidas y costosas que por lo general, no suelen ser de mucha capacidad si se compara con las memorias DDR que son más comunes. En el FPGA de número de parte XC7Z010-1CLG400C, cada uno de los bloques son de 36 KB, y pueden ser configuradas en dos modos: *single port BRAM* de 36 KB cada bloque y *dual port BRAM* con 18 KB cada bloque. En la **Figura 2.10** se ejemplifica la estructura de estos dos modelos.

En cada uno de los bloques las operaciones de lectura y escritura están sincronizadas, sin embargo, cada uno de los puertos es independiente con respecto al otro, solo compartiendo la información que tiene almacenada.

2.6.4. *Application Processing Unit* (APU)

Es la parte del FPGA que se encuentra localizada dentro del PS (*Processing System*) [33], y provee soporte computacional de propósito general empleando lenguajes de programación estándar como C o Java. En el caso del FPGA Zybo Zynq XC7Z010-1CLG400C que se está proponiendo, contiene un procesador ARM-Cortex A9.



(a) BRAM de un solo puerto.

(b) BRAM de dos puertos.

Figura 2.10: Configuración de las BRAM Single y Dual. El nombre de los puertos mostrados en la figura son los siguientes: *clk* es el reloj, *ena* activa la escritura, *addr* indica la dirección de lectura y escritura, y por último *din* y *dout* es el datos que está siendo escrito o leído.

La APU es el componente más importante ya que está en constante interacción con el PS, el PL (*Programmable Logic*) que contiene las IP's, y cada uno de los componentes individuales que contiene el dispositivo como pueden ser los puertos y las memorias. Este permite tener un control sobre todo el dispositivo.

Capítulo 3

Desarrollo de actividades

3.1. Introducción al problema

Los algoritmos que son empleados para el procesamiento de imágenes suelen ser demandantes en coste computacional, debido a que hay que procesador cada uno de los píxeles de toda la matriz correspondiente a una imagen. Se han elaborado diferentes implementaciones a nivel de software para aumentar la velocidad de procesamiento empleando diferentes técnicas como la que podría ser algoritmos paralelos, algoritmos heurístico o aprovechando el poder de *hardware* que ofrece las tarjetas gráficas. Los algoritmos de visión estéreo no son la excepción, ya que la búsqueda de la correspondencia que se hace en un par de imágenes conlleva que, una misma operación de estimación de coste tenga que ser repetida múltiples veces.

Ahora, el problema que conlleva esta investigación requiere que la implementación de estos algoritmos de correlación sea a nivel de *hardware*, esto es prácticamente posible pero esto acarrea un problema; debido a se busca crear un sistema en tiempo real, es necesario entonces aprovechar las capacidad del hardware para implementar algunos proceso en paralelo, pero debido a las capacidades limitadas que el dispositivo tiene hace que se tenga que reducir la precisión de la información obtenida con el beneficio de que los tiempos de procesamiento se ven reducidos. Además, algunos filtros para mejorar los resultados finales también pueden ser implementados. Entre los trabajos revisados, se han propuesto múltiples maneras en que se pueden realizar la implementación y los algoritmos a utilizar [34], [7], [35], [18], [36], [25], [37].

3.2. Obtención de los componentes

El dispositivo FPGA que se propone para usar es SoC Xilinx Zynq-7000 (**Figura 3.1**) disponible en la institución. La razón que se decidió emplear este dispositivo es por los costes que están teniendo actualmente en el mercado, además de que es suficiente para los objetivos de este trabajo.

Las especificaciones de la FPGA que se va a emplear se pueden observar en la **Tabla 3.1** [38]. Además, el modelo de FPGA que se está proponiendo es una versión anterior a las que se encuentras actualmente, esto significa que tiene un puerto VGA, a diferencia de los actuales que tienen dos puertos HDMI y un puerto Pcam.

Número de parte	XC7Z010-1CLG400C
Slices lógicos	4,400
6-input LUTs	17,600
Flip-Flops	35,200
BRAM	270 KB
DSP *Slices*	80
Recursos del reloj	*Zynq PLL with 4 output, 2 PLLs, 2 MMCMs, 125 MHz external clock.*
ADC Interno	*Dual-channel, 1 MSPS.*

Tabla 3.1: Especificaciones clave del FGPA propuesto para el proyecto.

3.3. Selección del *software*

En el desarrollo de este proyecto, se van a emplear diferentes programas y librerías para el diseño de la metodología, diseñar los componentes y hacer las comparaciones de resultados. Algunos de estos programas exigen gran cantidad de recursos de hardware de la PC, así que algunas partes del proyecto solo fue posible trabajarlas en la misma institución. Entre los programas empleados para este proyecto se mencionan a continuación:

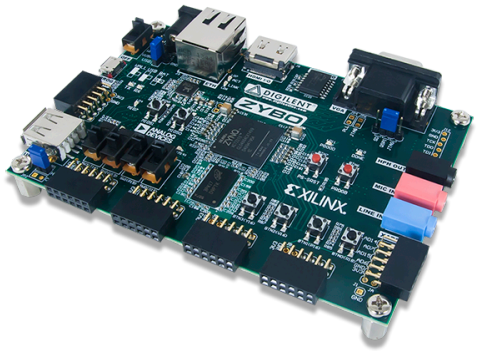


Figura 3.1: SoC Xilinx Zynq-7000 empleado para este proyecto.

- **Xilinx Vivado**. Programa principal en el que se diseñan cada uno de los componentes que están contenidos en el PL del FPGA, así como las conexiones que tienen entre si.
- **Xilinx Vitis**. En este se hace el código escrito en C que se asigna al microcontrolador. Requiere que se asigne el hardware diseñado en Vivado como plataforma del proyecto.
- **Visual Studio Community 2022**. *Software* por parte de Microsoft en el que se pueden diseñar aplicaciones en diferentes lenguajes. Para este proyecto se usaron los lenguajes C, C++ y C# para diseñar aplicaciones de prueba de los algoritmos, así como también fue aquí donde se utilizo la librería de OpenCV para hacer las comparaciones.
- **Visual Studio Code**. *Software* por parte de Microsoft que hace de editor de texto con soporte a diferentes lenguajes de programación. Las aplicaciones de consola en C se desarrollaban empleando este programa y la compilación se hacían con el compilador de MinGW 6.3.0.
- **Active-HDL Student Edition**. Programa utilizado para hacer las simulaciones de los componentes diseñados en VHDL. Debido a que se está trabajando con hardware, es difícil saber como se comportan los registros, así que por medio de este programa se tiene en tiempo real, un aproximado del comportamiento de cada uno de los componentes.
- **R**. Es un interprete que permite hacer operaciones matemáticas.
- **OpenCV**. Librería que tiene múltiples funciones que permiten la carga y el procesado de imágenes.

Algunos de estos programas tienen versión libre, es decir, que se pueden instalar y usar, mientras que otros requieren un registro antes de emplear. El *software* que ofrece Xilinx es el que exige más recursos de la PC, tanto como disco duro como memoria RAM, y solo se instaló en equipos de la institución. El programa de Active-HDL Student Edition es el único que requiere una licencia, por lo que solo se empleo el tiempo que la licencia gratuita otorgaba.

3.4. Diseño de los algoritmos

Se mencionaba anteriormente que la metodología va a estar dividida en dos partes: los componentes que van a ir en el FPGA y la otra parte que va a ir en el microcontrolador. En la parte del microcontrolador es donde se hace la carga de las imágenes y se aplica el preprocesado correspondiente. En la **Figura 3.2** se muestra la secuencia que van a tener los algoritmos en el sistema, fue diseñado basado en los trabajos propuestos por [39] y [7].

Figura 3.2: Diagrama general de la metodología propuesta. El proceso marcado con rojo es el implementado en el FPGA, mientras que en azul es la parte que está en el microcontrolador.

3.4.1. Obtención de la información

No se toma la directamente la imagen completa, ya que tomar en cuenta toda la información que contiene las cabeceras y los bits de sincronía es un gasto innecesario de tiempo, por lo que se optó por tomar solo el arreglo que contiene los píxeles. En la **Sección 2.2** se menciona la estructura que tienen las imágenes de formato *.bmp*, por lo que a partir de la dirección 54_d hasta el final del archivo contiene la información relacionada con los píxeles. Se diseñó un programa en C++ que hace la lectura de las imágenes *.bmp*, además de diseñar una función que permite extraer esta información y almacenarla en un archivo de texto. En ese código se definió una función llamada *save()* que se muestra a continuación:

```
1 bool save(const char *_filename, img_format _format) {
2     switch (_format) {
```

```
        case img_format::BMP: return save_bmp(_filename);
        case img_format::BIT: return save_bit(_filename);
        default: return false;
    }
}
```

Además de recibir el nombre del archivo con el que se va a guardar, también se tiene que pasar como parámetros el formato en el que lo va a hacer. Hay dos opciones: el primero es *.bmp* que guarda la información como una imagen y la otra es *.bit* que almacena el arreglo de los píxeles como un archivo de texto.

Las imágenes con la que se van a estar trabajando son de 320×240 píxeles de un arreglo total de tamaño $230,400$ bytes. En esta sección todavía se consideran imágenes de 24 bits, así que un filtro de conversión a escala de grises es necesario antes de continuar.

3.5. Diseño del hardware

Cada uno de los algoritmos vistos con anterioridad ahora deben de ser implementados en el hardware. Sin embargo, debido a las limitantes de los recursos del FPGA es necesario dividir el problema en dos: los algoritmos que van a ser implementados a nivel de hardware en el FPGA, y aquellos que van a ser implementados en el microcontrolador. Como la velocidad del microcontrolador es inferior al del FPGA, entonces aquellos algoritmos que demanden más coste computacional pueden ser implementados ahí, mientras que aquellos que no sean tan demandantes pueden ser implementados en el microcontrolador. El algoritmo de correlación va a ser colocado en la parte del PL del FPGA debido a que es el más demandante.

3.5.1. Problemas encontrados durante el diseño

Durante el desarrollo de este sistema, se encontraron algunos inconvenientes que no habían sido previstos desde el principio, por lo que requirió de un replanteamiento de la lógica del diseño y los resultados finales. Para que estos no sean un problema a futuro, se hace mención de estos a continuación.

Espacio en la BRAM

El FPGA solo dispone de $2,211,840$ bits de BRAM, por lo que no es posible almacenar las tres imágenes enteras en los bloques de memoria ya que también se tiene que reservar recursos para el resto de el sistema. La solución fue dividir la imagen en dos partes. Como las dimensiones propuestas de la imagen son de 320×240, y se están implementando ventanas de 3×3, las dimensiones de cada una de las partes son de 320×123, sin embargo, esto significaría que la imagen se tiene que procesar en dos partes y producir una única salida. Cada una de las partes tienen un total de $39,360$ píxeles de 8 bits cada una.

Cantidad de LUT's empleadas

Este FPGA tiene disponible $17,600$ LUT's (*Lock-Up Tables*) y cada uno de los registros definidos a lo largo de el proyecto emplea cierta cantidad de estos. La metodología que se está siguiente propone una lógica en paralelo, es necesario que cada uno de los procesadores tenga asignada cierta cantidad de LUT's, por lo que si el tamaño de las ventanas de correlación son demasiadas grandes con un número de procesadores de 64, entonces el consumo de este recurso va por más del $120\,\%$ con ventanas de 5×5. Así que la solución a esto fue reducir la ventana a 3×3 con un uso de LUT's del $83\,\%$.

Inclusión de las librerías en el microcontrolador

Dentro del código en C que está incluido en el microcontrolador, es necesario especificar en el archivo *Makefile* que se van a emplear las librerías nativas de C. Estas librerías son usadas dentro del código para hacer algunas operaciones de desplegado de información, sin embargo, estas no están contempladas para la compilación generando problemas.

El archivo que se tiene que modificar es nombrado como *objects.mk* y se encuentra en la carpeta *Debug* de el proyecto. El contenido del archivo *Makefile* original es el que se muestra a continuación:

```
1 USER_OBJS :=
2
```

```
3 LIBS := -Wl,--start-group,-lxil,-lgcc,-lc,--end-group -Wl,--start-group,-
    lxilffs,-lxil,-lgcc,-lc,--end-group -Wl,--start-group,-lrsa,-lxil,-
    lgcc,-lc,--end-group
```

este debe de ser reemplazado por las siguientes lineas:

```
1 USER_OBJS :=
2
3 LIBS := -Wl,--start-group,-lxil,-lgcc,-lc,-lm,--end-group -Wl,--start-
    group,-lxilffs,-lxil,-lgcc,-lc,-lm,--end-group -Wl,--start-group,-lrsa
    ,-lxil,-lgcc,-lc,-lm,--end-group
```

o también solo se puede agregar ***-lm*** después de cada ***-lc*** separado por una coma. Cabe mencionar que este archivo se tiene que cambiar cada vez que se va a iniciar el proceso de compilación.

Controlador de la IP AXI

Después de la creación de la IP con el protocolo AXI, es necesario cambiar el contenido de archivo *Makefile* contenido en el *driver* de la IP. Esto es debido a que Vivado emplea una versión anterior del compilador MinGW mientras que Vitis no, así que esto crea conflictos a la hora de compilar el proyecto. El contenido del archivo *Makefile* empleado en este proyecto, se puede observar en el **Anexo C**.

3.5.2. Diseño de los componentes

El diseño del hardware en el FPGA se hace mediante componentes, es decir, crear bloques que tengan funciones especifica y por medio de los puertos hacer que trabajen en conjunto. En el programa Vivado, el diseño del *hardware* es con componentes que ya están por defecto en la aplicación, y que incluyen las funciones principales para lograr la comunicación entre los demás componentes. Sin embargo, en la misma aplicación, se pueden crear componentes personalizados, o *IP's*.

En este proyecto se creo una IP AXI que va a hacer de interfaz entre el FPGA y el microcontrolador, y se va a encargar de enviar los datos y controlar los procesos mediante registros. Los registros de este proyecto se pueden visualizar en la **Figura 3.3**. Se definieron 6 registro de

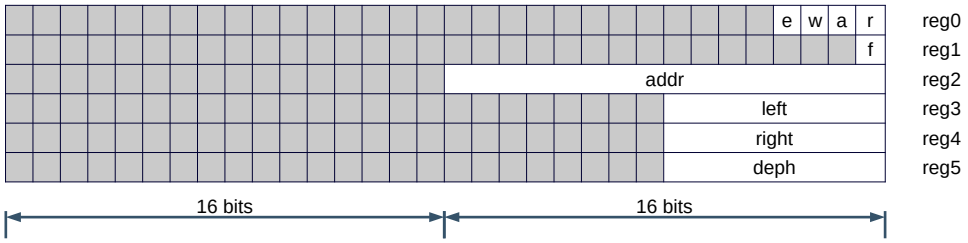


Figura 3.3: Los registros definidos para este proyecto en el controlador AXI. Cada una de las celdas corresponderían a cada uno de los bits, siendo de un total de 32 bits cada registro, y las áreas sombreadas son aquellos bits que no se están empleando.

32 bits cada uno (como mínimo se pueden crear 4 registro de 32 bits cada uno), de los cuales 4 registros son de entrada y 2 de salida. En el primer registro se pasan las señales de control de el sistema: **r** para la señal de reinicio (*rst*), **a** para la señal de activación (*adv*), **w** para activar la escritura (*wea*) y **e** para activar la lectura (*ena*). En el segundo registro se recibe la señal **f** de finalizado (*finished*). En el tercer registro se manda la dirección de lectura y escritura de 16 bits. Por último en los tres registros restantes se mandan los píxeles de las imágenes; un registro por cada píxel de 8 bits por imagen.

Dentro de el controlador AXI creado, es donde se definen los componentes creados para este proyecto. Cada uno de estos componentes es descrito en las siguientes secciones.

Librería *constants.vhd*

Entre todos los componentes definido, se emplean algunos registros con valores constantes y ayudan a especificar las propiedades de la imagen, así como también algunos parámetros que permiten controlar los procesos. Es posible definir algunos de estos parámetros como constantes empleando un *package*, que pueden ser empleados entre los diferentes componentes a lo largo del hardware. Cada uno de los registros que contiene este *package* se definen a continuación:

- ***SIZE***. Tamaño total del vector que contiene todos los niveles de gris de la imagen. Es un equivalente a guardar todos los píxeles de la imagen en un solo vector, en el que su tamaño total es $w \times h$. Tiene un valor constante de 39,360.
- ***D***. Número total de niveles de disparidad en los que se va a hacer la búsqueda de la correspondencia. Tiene un valor por defecto de 64.

- ***W***. Tamaño de las ventanas que se usan para hacer las búsquedas de la correspondencia en el operador local. Las ventanas son de 3×3.

- ***WW***. Número total de elementos que tienen cada una de las ventanas. Es el resultado de hacer la multiplicación $3 \times 3 = 9$ elementos.

- ***CONST_DATA_W***. Anchura en bits que puede tomar los valores de los píxeles. Como estos valores están dentro del intervalo de 0 a 255 niveles de gris, estos se pueden almacenar en 8 bits.

- ***CONST_ADDR_W***. Anchura de bits que puede tomar los valores para almacenar una dirección de memoria. Como la imagen tiene un tamaño total de 39,360, el intervalo de direcciones que puede tomar es de 0 a 39,359, así que esto se puede almacenar con 16 bits.

- ***CONST_COST_W***. Ancho de bits para la estimación de los costes. Hay que tomar en cuenta el máximo total que se puede almacenar en registros de este tamaño, el cual es equivalente a tomar el máximo de 8 bits (255) y multiplicarlo por el número de elementos a sumar (el número de elementos de la ventana es de 9), así que el total máximo es equivalente a 2,295. Este valor puede ser almacenado en registros de 12 bits.

- ***CONST_COOR_W***. Anchura que puede tomar una coordenada de la imagen. Las imágenes tiene un tamaño de 320×240, así que el valor máximo que puede tomar una coordenada es de 319, así que puede ser almacenado en registros de 9 bits.

- ***RES_X***. Registro que contiene el número de columnas de la imagen, el cual tiene un valor constante $0101000000_b = 320_d$.

- ***RES_Y***. Registro que contiene el número de filas de la imagen, el cual tiene un valor constante $001111011_b = 123_d$.

- ***MAX_X***. Valor máximo horizontal en el que se hace la búsqueda. Este es equivalente a la anchura $width - W$, donde W es el tamaño de la ventana de búsqueda. Esto se hace para mantener la ventana de búsqueda dentro de la imagen. Este registro tiene un valor constante de $100111100_b = 316_d$.

- ***MAX_Y***. Valor máximo vertical en el que se hace la búsqueda. Equivalente al caso anterior, también se toma en cuenta la ventana con la que se está haciendo la búsqueda. El registro tiene como valor $001110111_b = 119_d$.

- ***MAX_W***. Número total de columnas que se analizan tomando en cuenta el tamaño de la ventana y los diferentes niveles. Su valor constante es $001000001_b = 65_d$.

- ***MAX_V***. Desde que el procesador principal tiene que hacer una espera mientras que la información es escrita en los registro (ver **Sección 3.5.2**), este valor es el número de ciclos de reloj que tiene que esperar. Su valor es equivalente a el número de columnas totales multiplicado por el número de filas. El valor con el que se define es de $011000101_b = 197_d$.

- ***MAX_I***. Tamaño máximo que tiene la ventana de búsqueda. Las ventanas que se emplean son cuadradas, así que tienen el mismo número de filas y columnas. Su valor constante es $0000000010_b = 2_d$.

- ***PIXELES***. Definición de un vector que contiene información de los píxeles leídos para su asignación a cada uno de los procesadores (ver **Sección 3.5.2**), cada uno de los elementos es de 8 bits de anchura.

- ***IMAGE***. Definición de un vector que contiene todos los niveles de gris de la imagen. El vector es de tamaño *size* y cada uno de los elementos es de 8 bits.

- ***COSTES***. Definición de un vector que contiene todos los costes estimados con la función SAD. El vector tiene un tamaño equivalente al número de niveles de disparidad, y la profundidad de bits es de 12.

Componente *match_control.vhd*

Este es el componente que engloba los demás componente. su principal función es establecer conexión con los puertos del controlador AXI a los procesadores internos, así como también hacer la lectura y escritura de la información de los píxeles en los bloques de memoria. La estructura del sistema principal se puede observar en la **Figura 3.4**, donde se puede observar las interacciones que tienen los componentes mas internos.

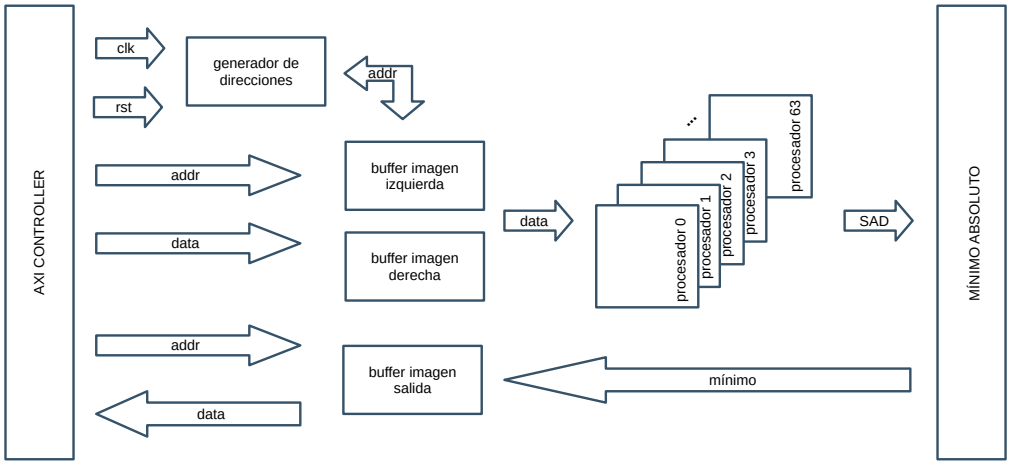


Figura 3.4: En la figura se puede observar las conexiones internas que tienen cada uno de los componentes.

La configuración de puertos para este componente se menciona a continuación:

- ***clk***. Puerto de el reloj principal de el sistema. Todos los demás componentes están sincronizados a este reloj.
- ***rst***. Puerto de la señal de reinicio. Esta señal es mandada por el microcontrolador cada vez que se va a enviar un reinicio de datos. Todos los demás componentes están conectados a esta señal.
- ***adv***. Puerto de la señal que activa el sistema. Como todos los componentes están activos, es necesario desactivar el componente principal de procesamiento mientras se esté leyendo o escribiendo la información.
- ***wea***. Puerto de la señal que activa la escritura en la BRAM. Mientras este puerto esté en 0, no se va a hacer ninguna operación de escritura hasta que cambie a 1.
- ***ena***. Puerto de la señal de lectura. Con esta señal se permite la lectura de la información una vez que el proceso haya terminado, mientras esté en 0 no se puede hacer lectura.
- ***finished***. Puerto de la señal de finalizado. Esta señal marca al microcontrolador para indicar que el proceso ya terminó y es necesario un nuevo set de datos.
- ***addr[15...0]***. Puerto de señal donde se manda la dirección. Esta dirección puede ser empleado para escribir en los *bufferes* izquierdo y derecho, así como también para leer el

buffer de salida. La lectura o escritura se controla mediante la señales *ena* o *wea*.

- ***left[7...0]***. Puerto de la señal donde se transmite el píxel de la imagen izquierda. Este píxel se escribe en el *buffer* izquierdo en la dirección mandada en el puerto *addr*.
- ***right[7...0]***. Puerto de la señal donde se transmite el píxel de la imagen derecha. Este píxel se escribe en el *buffer* derecho en la dirección mandada en el puerto *addr*.
- ***deph[7...0]***. Puerto de la señal donde se transmite el píxel de la imagen resultante de la correlación. Este píxel se lee de el *buffer* resultante (llamado en este proyecto *deph*) en la dirección mandada en el puerto *addr*.

Componente *image_memory.vhd*

Como se había mencionado anteriormente, el FPGA solo dispone de 2.1 Mb de BRAM, distribuidos en 60 bloques de 36 Kb cada uno [40]. Para almacenar una parte de la imagen se hace empleo de estos bloques de memoria, definiendo un generador. El propio *software* de Vivado contiene componentes que se pueden emplear como generadores y son configurables, sin embargo, para este caso es necesario crear un componente especifico para almacenar una imagen. Dentro de la documentación de Xilinx provee información de como crear un generador de bloques de memoria, la cual puede ser empleado para almacenar las imágenes que se necesiten.

En la **Sección 3.5.1** se resalta el problema encontrado relacionado con el espacio en la memoria de el FPGA. La solución a esto fue que la imagen se tuvo que partir en varias partes y realizar el procesamiento de este modo. El microcontrolador estaría enviando cada parte de la imagen cada vez que se transmite una señal de activación por medio del puerto *finished*. La imagen quedó segmentada en dos partes de 320×123 en escala de grises. Como cada parte tiene un total de $320 \times 123 = 39,360$ bytes, es posible crear un generador que contenga un vector de tal dimensión. Además, el bloque creado es simple con dos puertos para la lectura y escritura respectivamente y compartiendo un único reloj. Este componente es necesario crearlo tres veces, uno por cada imagen que se está procesando. La configuración de los puertos de este generador se especifica a continuación:

- ***clk***. Puerto de la señal de reloj principal con el que está sincronizado.

- ***wea***. Puerto de la señal que habilita la escritura de la información.
- ***ena***. Puerto de la señal que habilita la lectura de la información.
- ***waddr[15...0]***. Puerto por donde se transmite la dirección de escritura.
- ***raddr[15...0]***. Puerto por donde transmite la dirección de lectura.
- ***din[7...0]***. Puerto por donde se transmite la información que se va a escribir en los registros.
- ***dout[7...0]***. Puerto de salida por donde se transmite la información leída.

Las señales de escritura solo se activan desde el microcontrolador cuando este hace la transmisión de la información de la DDR a la BRAM, mientras que la señal de lectura es habilitado dentro de los procesos para recuperar la información. Para la imagen de salida el proceso es inverso; la señal de escritura se activa dentro de los procesos para escribir los resultados y la señal de lectura se activa desde el microcontrolador para transmitir el resultado de la BRAM a la DDR.

Componente *main_processor.vhd*

Los componentes que se encargan de los procesos principales están englobados dentro de este. La información transita desde los componentes externos donde está almacenada la información y es distribuida entre los procesadores que son los que producen la información de salida. Los puertos de este componente son los siguientes:

- ***clk***. Puerto por donde se conecta el reloj principal.
- ***rst***. Puerto por donde se conecta y recibe la señal de reinicio del sistema.
- ***adv***. Puerto por donde se conecta la señal recibida desde el microcontrolador para activar el procesado una vez que termine la escritura.
- ***finished***. Puerto por donde se conecta la señal de salida desde el procesador para indicar que se terminó una parte. Este puerto está conectado directamente a la señal *vs* vista más adelante (ver **Sección 3.5.2**).

- ***addr_left[15...0]***. Puerto de salida por donde se transmite la dirección de lectura en la memoria de la imagen izquierda. De 16 bits.
- ***addr_right[15...0]***. Puerto de salida por donde se transmite la dirección de lectura en la memoria de la imagen derecha. De 16 bits.
- ***addr_out[15...0]***. Puerto de salida por donde se transmite la dirección de escritura en la memoria de la imagen de salida. De 16 bits.
- ***pixel_left[7...0]***. Puerto de entrada donde se transmite los píxeles de la imagen izquierda.
- ***pixel_right[7...0]***. Puerto de entrada donde se transmite los píxeles leídos de la imagen derecha.
- ***pixel_out[7...0]***. Puerto de salida por donde se transmite los píxeles de escritura de la imagen de salida.
- ***wea***. Puerto de salida para activar la escritura de la información.
- ***ena***. Puerto de salida para activa la lectura de la información.

Los componentes definidos dentro de este son: un generador de direcciones, 64 procesadores (uno por cada nivel) encargados de procesador las ventanas y producir una salida, y un componente para calcular el mínimo absoluto.

El generador de direcciones se encarga de hacer los recorridos de la imagen, ademas de producir la columna y fila que se está leyendo. Por cada iteración producida por este generador, se generan direcciones de lectura en la imagen izquierda y derecha, además de una dirección de escritura para la imagen resultante. Por cada dirección producida, se obtiene un píxel que es distribuido entre los procesadores. Por cada píxel izquierdo es distribuido directamente a cada uno de los 64 procesadores, mientras que para los píxeles de la imagen derecha la asignación es diferente. Se emplea un registro que hace de conexión con los 64 procesadores, por donde se transmite el píxel leído de la imagen derecha, en el código se emplea una estructura *case* usando la columna producida por el generador de direcciones definido anteriormente. Dependiendo de

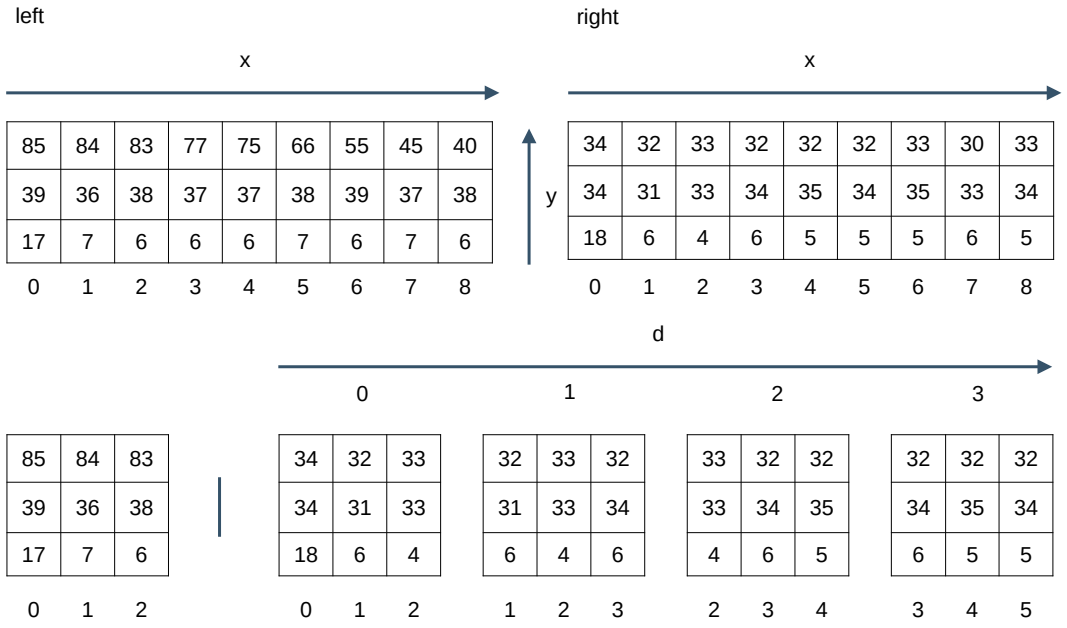


Figura 3.5: Proceso en que se muestra el procedimiento para hacer la escritura de los datos en cada uno de los procesadores. Para la creación de la ventana todos los datos leídos de la imagen izquierda se almacenan directamente en el *buffer*, mientras que para la creación de las ventanas para la imagen derecha se va almacenando la información conforme a la columna leída enumerada en esta figura.

la columna en la que está posicionado, el píxel entra en los diferentes procesadores, además de activar la escritura dependiendo de si este píxel es asignado o no al procesador, o en otras palabras, los píxeles de la columna 0 entran en el procesador 0, los píxeles de la columna 1 entran en los procesadores 0 y 1, tal y como queda ilustrado en la **Figura 3.5**. Además, también se encarga de activar la escritura para el píxel izquierdo en las primeras 3 columnas. El número total de columnas que se leen son de 65, esto es por el hecho de que lee desde 0 a 63 por niveles, además de que se toma en cuenta 2 columnas más por la ventana. Por cada columna, se leen 3 píxeles, uno por cada fila.

Por último, las salidas de cada uno de los procesadores es almacenado en un vector que contiene la información de costes de 16 bits obtenidos mediante la función de estimación de la correspondencia. Este vector es asignado al componente *disparities* que va a hacer la búsqueda de el mínimo absoluto de el vector de entrada, y el resultado corresponde a un nuevo nivel de gris. Este píxel nuevo se escribe en la dirección obtenido mediante el generador de direcciones.

En este componente también se definen registro para controlar el sistema. Cada uno de los procesadores tienen una secuencia de estados (definidos en la **Sección 3.5.2**). Cada uno de los procesadores produce una señal indicando que está a la espera de nuevos datos, esta señal se combina con la señal de entrada por el puerto *adv* para activar el generador de direcciones, mientras uno de estos puertos esté desactivado, entonces el generador almacena la posición actual y está a la espera de una activación. Cada uno de los procesadores tienen que producir esta espera mientras que se está haciendo la operación y acumulación de los costes.

Finalmente, se utiliza la señal transmitida por el puerto *vs* de el generador de direcciones, para indicar que se termino de recorrer el arreglo de la imagen y es conectado al puerto *finished* para indicar al microcontrolador que se terminó el proceso.

Componente *addr_generator.vhd*

Este es el componente que se encarga de hacer los recorridos por todos los píxeles de las imágenes. Este produce como salida una dirección de memoria en donde se almacena el píxel de la coordenada correspondiente, ademas de también producir señales que controlan el estado de los recorridos de la imagen. Los puertos definidos para este componente son los siguientes:

- ***clk***. Entrada del reloj principal del sistema.
- ***rst***. Entrada de la señal de reinicio del sistema, mandada por el microcontrolador.
- ***adv***. Señal de activación del sistema.
- ***vs***. Señal de sincronía vertical. Cuando esta señal se activa indica que se termino una imagen completa.
- ***column[8...0]***. Puerto por donde se transmite la columna en la que actualmente se está leyendo. Es diferente a el valor de la coordenada x.
- ***row[8...0]***. Puerto por donde se transmite la fila que se está leyendo. Por cada columna, se lee hasta un máximo de W filas.

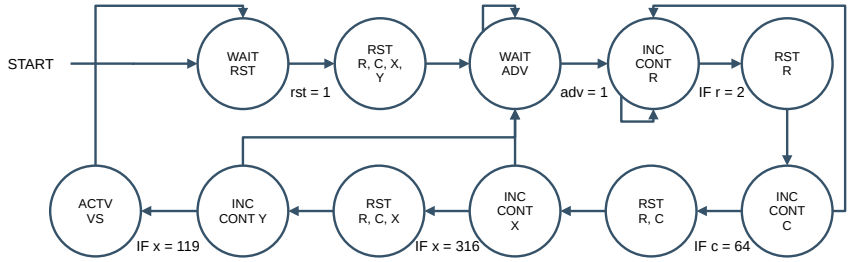


Figura 3.6: Secuencia de estados del componente *addr_generator* en la que se ejemplifica la lógica secuencial de este componente.

- ***addr_left[15...0]***. Puerto por donde se transmite la dirección de lectura del píxel de la imagen izquierda.
- ***addr_right[15...0]***. Puerto por donde se trasmite la dirección de lectura del píxel de la imagen derecha.
- ***addr_out[15...0]***. Puerto por donde se trasmite la dirección de escritura del píxel de la imagen de salida.

Como se requiere trabajar con ventanas, es más sencillo crear las ventanas empleando el sistema de coordenadas x, y, por lo que este componente maneja un contador por cada coordenada. Estos registros fueron llamados sencillamente x y y para su fácil manejo. También se maneja un par de registro para hacer el conteo de las filas y las columnas, llamados c para el contador de columnas, y r para el contador de filas, todos estos registros se inicializan a sus valores iniciales de ceros. La diferencia de estos registros radica en que los contadores x y y son usados para hacer los recorridos por la imagen, mientras que los contadores c y r son para los recorridos de la ventana. Estos contadores se combinan empleando el componente *get_addr*, así que dentro se crean tres instancias; una por cada imagen. Para producir la dirección de la imagen izquierda y derecha se hace la suma entre los contadores $x + c$ y $y + r$, y el resultado de esta suma es la entrada a estos componentes. Para las direcciones de la imagen de salida solo se transmite los contadores x y y. La dirección producida por cada uno de estos componentes es conectado a los puertos de salida por donde se transmiten las direcciones. Y por último se define un registro para controlar cuando se terminó de recorrer una imagen, llamado *this_vs* con un valor inicial de *0*.

Los contadores son cambiados mediante un proceso interno. Este proceso cuenta con dos

estados: cuando se recibe una señal de *reset* y cuando se recibe una señal de reloj. Cuando se recibe la señal de *reset* o reinicio los contadores y el registro *vs* son llevados a sus valores iniciales. Cuando se recibe una señal de reloj, es necesario incrementar los contadores, pero estos se hacen de la siguiente manera. Verifica primero si la señal *adv* está activa, mientras que este inactiva solo mantiene los registros en su estado actual, cuando se activa entonces procede a incrementar el contador r, cuando este llega a su valor máximo $r = MAX\_I$ entonces se reinicia el registro r e incrementa el contador c, cuando c llega a su máximo valor $c = MAX\_W$ entonces entra a un estado de espera e incrementa el contador de x, cuando x llega a su máximo valor $x = MAX\_X$, entonces incrementa el contador y y reinicia los contadores x, c y r. Cuando y llega al su máximo valor $y = MAX\_Y$ entonces activa el registro *this_vs* indicando que se termino de procesar la parte de la imagen. Este proceso queda ejemplificado en el diagrama de estados que se puede visualizar en la **Figura 3.6**. Cada vez que el registro c llega a su máximo valor, este proceso entra en un estado de espera hasta que los procesadores terminen de procesar la información.

Los registros c y r están conectados a los puertos *column* y *row* respectivamente, y el registro *this_vs* está conectado a el puerto *vs*.

Componente *get_addr.vhd*

Los recorridos de las imágenes se hacen por medio de dos contadores como se mencionaba anteriormente. Sin embargo, como el modo en el que se estructura la imagen es como un arreglo, es necesario convertir tales contadores en una dirección de memoria para leer el píxel correspondiente en el arreglo. El par de coordenadas se convierte en una dirección siguiendo la operación $addr = (y \times width) + x$ donde x y y correspondería a los pares ordenados, mientras que $width$ equivale al ancho de la imagen. En este componente se definen los siguientes puertos:

- ***x[8...0]***. Entrada de 9 bits de ancho, es el valor de la coordenada x.
- ***y[8...]***. Entrada de 9 bits de ancho, es el valor de la coordenada y.
- ***addr[15...0]***. Salida de 16 bits de ancho, contiene la dirección de memoria calculada.

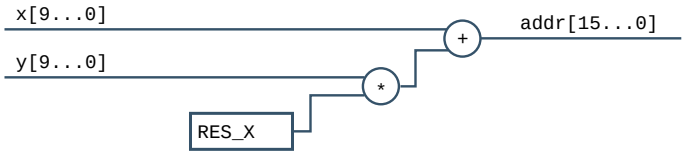


Figura 3.7: En la figura se muestra el proceso en el que se realiza la operación para obtener una dirección de memoria. El registro *RES_X* es una constante que contiene el ancho máximo de la imagen definido anteriormente en el *package*.

En la **Figura 3.7** se observa como está estructurado el componentes. Este solo aplica una operación de combinatoria, es decir, que las entradas por los puertos se operan para producir una única salida.

Componente *processor_buffer.vhd*

Este es el componente que se encarga de hacer las operaciones esenciales en la búsqueda de correspondencia. Desde que se tiene que hacer la búsqueda en 64 niveles, es necesario crear uno de estos procesadores por cada nivel para hacer la búsqueda aprovechando el paralelismo. Los puertos que tiene este componente se definen a continuación:

- ***clk***. Entrada del reloj principal de el sistema.
- ***rst***. Entrada de la señal de reinicio de el sistema, mandada por el microcontrolador.
- ***adv***. Señal de activación del sistema.
- ***wrenal_l***. Puerto por donde se transmite la activación de la escritura en los registros de los píxeles de la ventana de la imagen izquierda.
- ***wrenal_r***. Puerto por donde se transmite la activación de la escritura en los registro de los píxeles de la ventana de la imagen derecha.
- ***left[7...0]***. Puerto por donde se hace la transmisión del píxel de la imagen izquierda.
- ***right[7...0]***. Puerto por donde se hace la transmisión del píxel de la imagen derecha.
- ***row[8...0]***. Puerto por donde se transmite la fila donde se va a escribir el píxel de entrada.
- ***cost[11...0]***. Puerto por donde se transmite el resultado obtenido.

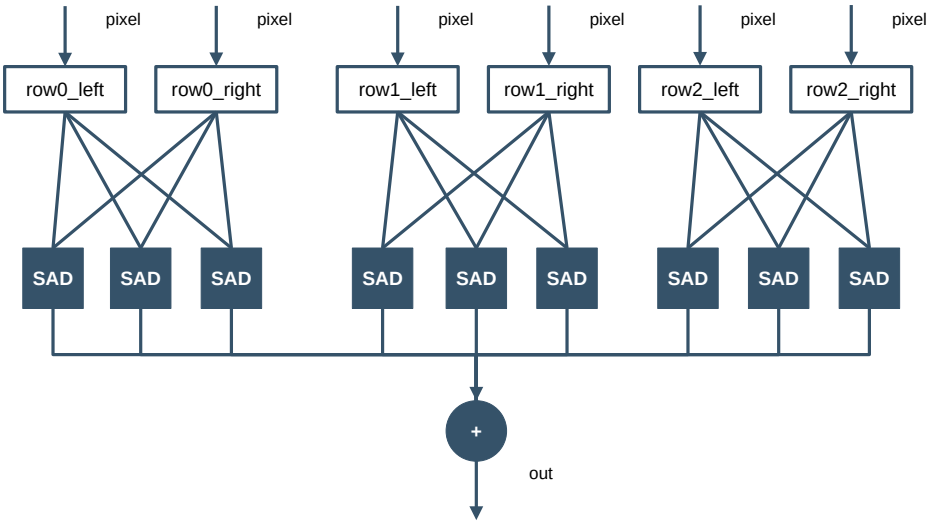


Figura 3.8: Representación de cada uno de los procesadores de ventanas. La información escrita en cada registro es una entrada a cada uno de los procesadores de la función SAD, y sus salidas son acumuladas para producir una única salida.

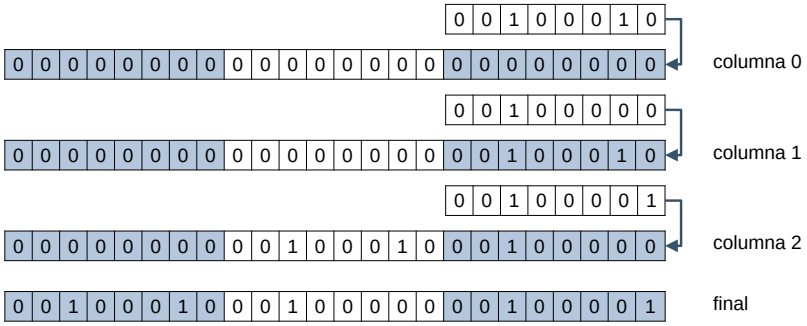


Figura 3.9: Representación del proceso de escritura de la información en los registros en cada procesador. Cada uno de los registros de 24 bits se recorre para que el nuevo píxel de 8 bits se escriba al final del vector.

- ***process_wait***. Salida de la señal de espera de el procesador.

Los componentes incluidos son: el componente *sad* que hace la operación de obtener el coste empleando los píxeles de ambas ventanas, el componente *acumulator* que hace la acumulación de las salidas del componente anterior, y finalmente un componente auxiliar *flip_flop* que almacena el resultado hasta la siguiente iteración. En la **Figura 3.8** se muestra como están interconectados todos los componentes dentro de este procesador.

Desde que se tiene que cada ventana son de 3×3 y que cada píxel es de 8 bits, entonces

Nombre	Valor	Activación	Espera
ESCRITURA	00_b	001_b	1_b
OPERACIÓN	01_b	100_b	0_b
ACUMULACIÓN	10_b	010_b	0_b
NO OPERATIVO	11_b	000_b	0_b

Tabla 3.2: Definición de los estados que toma el componente *processor_buffer* y los valores que toma el registro *activate* por cada estado.

cada uno de los procesadores tienen 6 registros de 24 bits; uno por cada fila para los diferentes *buffers* de las ventanas izquierda y derecha. En la **Figura 3.9** se muestra el proceso de escritura de los píxeles en cada registro, en el que cada píxel nuevo que se recibe se escribe en la parte final del vector, y esto se hace por cada fila señalado por el puerto de entrada *row*. Así, de esta forma, no es necesario un nuevo puerto para indicar la columna, ya que se sabe de antemano que los últimos 8 bits corresponden a la última columna de esta ventana. El proceso de escritura es controlado por las señales de reloj, además que que por medio de las señales de activación *wrenal_l* y *wrena_r* se evita que se escriba información adicional y la escrita no se pierda.

Este componente tiene cuatro estados definidos en un registro de 2 bits ilustrados en la **Tabla 3.2**. Estos estados son para controlar las operaciones que están incluidas en este competente e indicar cuando es necesario esperar por nueva información. El incremento del contador es hecho por un proceso controlado por el reloj. El proceso es el siguiente: en el primer estado es de espera hasta que se manda una señal de activación por el puerto *adv*, en el siguiente estado es el de escritura y en este tiene que esperar hasta que todos los píxeles son escritos; el tiempo que espera es equivalente al número de filas y el de columnas que se leen. En el siguiente se espera hasta que se hace la operación de obtención de costos y se almacenan en un vector, y al final se activa el estado de acumulación en el que todos los elementos del vector se suman y se obtiene un total que es el resultado de salida.

En cada estado, se activan también diferentes opciones dependiendo de lo que se hace en dicho estado, esto también queda ilustrado en la **Tabla 3.2**. La activación se hace mediante un

registro llamado *activate*, de 3 bits; desde el bit más significativo, el primer bit es para reiniciar el acumulador, el segundo bit es para activar el proceso de acumulación y el último bit es para activar el *flip-flop* en donde se almacena el resultado hasta la siguiente iteración. También se tiene que activar el estado de espera; solo en el estado de escritura es cuando está activo, mientras que en los otros estados es cuando está inactivo.

Durante el proceso de acumulación, por cada valor del contador se asigna a la entrada del componente del acumulador cada uno de los 9 valores que tiene el arreglo dependiendo del valor de este contador, además de que a esta entrada se le concatena una cadena de 4 bits de ceros para completar el total de 12 bits. El resultado final espera al siguiente estado para ser almacenado en el *flip-flop* para hacer la búsqueda del mínimo entre todos los procesadores.

Componente *sad.vhd*

Esta es la función básica para la estimación de los costes. La ecuación se explicó en el capitulo anterior (**Ecuación 2.1**), sin embargo, en este componente no se necesita toda la ecuación si no más bien la operación que se encuentra dentro de la sumatoria. Este operador queda descrito como $cost = |left - right|$ donde $left$ y $right$ corresponden a los píxeles de las ventanas de las imágenes izquierda y derecha respectivamente, mientras que *cost* es el valor resultante de la operación. La configuración de los puertos de este componente es el siguiente:

- ***left[7...0]***. Entrada de 8 bits de ancho, es el valor del píxel de la ventana de la imagen izquierda.
- ***right[7...0]***. Entrada de 8 bits de ancho, es el valor del píxel de la ventana de la imagen derecha.
- ***cost[7...0]***. Salida de 8 bits de ancho correspondiente a el resultado de la operación.

Este componente hace la función de combinatoria entre las señales tal y como se puede observar en la **Figura 3.10**. Entre los datos transmitidos por los puertos *left* y *right* se aplica una sustracción y empleando la función *abs* incluida en la librería *ieee.numeric_std* es posible obtener el valor absoluto al resultado de la sustracción.

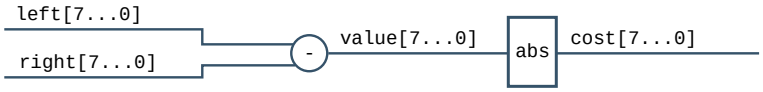


Figura 3.10: Componente *sad.vhd* que aplica el operador entre ambos píxeles.

Componente *acumulator.vhd*

Este componente hace la acumulación de cada uno de los nueve elementos contenidos en el arreglo temporal que almacena las salidas del operador SAD. La configuración de puertos es la siguiente:

- ***clk***. Entrada del reloj con el que se sincroniza el acumulador.
- ***rst***. Entrada de la señal de reinicio del acumulador.
- ***adv***. Señal de activación del acumulador.
- ***cost_in[11...0]***. Puerto por donde se transmite los datos de entrada almacenado en el vector.
- ***cost_out[11...0]***. Puerto por donde se transmite el resultado de acumulación.

Dentro de este componente está contenido un proceso que puede tomar dos estados diferentes. El primer estado es cuando se recibe la señal de reinicio (*rst*) y es cuando el acumulador se coloca en 0. El siguiente estado es cuando se recibe una señal de reloj, en este estado se verifica que este activo el acumulador mediante la señal del puerto *adv*, si está activa entonces el valor recibido por el puerto *cost_in* se suma con acumulador. El resultado de la acumulación es almacenado en el registro *this_acum* y que es asignado directamente a la salida por el puerto *cost_out*. Este registro es de 12 bits debido a que es el máximo valor que puede tomar la suma de los 9 elementos del arreglo, en el que cada elemento es como máximo de 8 bits.

Componente *flip_flop.vhd*

Desde que es necesario almacenar el resultado de la acumulación hasta que el proceso de buscar el mínimo absoluto se complete, se puede hacer con la ayuda de un componente auxiliar como un *flip-flop* para asegurar que la información resultante no se pierda. En este sistema está implementado un *flip_flop* tipo SR (*set-reset*), en el que se reciben dos señales: una para

reiniciar el valor almacenado, y uno para establecer un nuevo valor en el registro. Los puertos para este componente se definen a continuación:

- ***clk***. Entrada del reloj con el que se sincroniza.
- ***set***. Entrada de activación del *flip-flop*.
- ***rst***. Entrada de la señal de reinicio del *flip-flop*.
- ***din[11...0]***. Puerto por done se transmite el dato que se va a almacenar.
- ***dout[11...0]***. Puerto por donde se transmite el valor almacenado en el *flip-flop*.

Este *flip-flop* tiene dos estados: cuando se recibe una señal de reinicio por el puerto *rst*, en el que se elimina el valor almacenado, y cuando se recibe una señal de *set* en el que a la salida se asigna la entrada. Hay un tercer estado, cuando tanto la señal de *rst* y la señal de *set* son uno, este estado no es valido y se elimina el resultado produciendo una señal de alta impedancia.

Componente *disparities.vhd*

Cada uno de los procesadores produce una salida producto de la acumulación de los elementos del *buffer* temporal que también son almacenados en un *buffer* que es la entrada a este componente para hacer la búsqueda del mínimo absoluto. El mínimo absoluto consiste en buscar la posición (o índice) donde se encuentra localizado el valor mínimo del vector de entrada, y este se toma como nuevo nivel de gris del píxel para la imagen de salida.

Una manera de encontrar el mínimo es mediante una búsqueda secuencial en el que se hace la comparación de cada uno de los elementos, sin embargo, esto solo hace que el proceso sea más lento ya que cada comparación tomaría un ciclo de reloj y si tenemos 64 elementos entonces la búsqueda tomaría los 64 ciclos en terminar. Es posible aprovechar las capacidades del hardware para agilizar este proceso, y una manera es implementar la búsqueda mediante *arboles binarios*. En cada nivel del árbol se hace la comparación por pares que producirían una salida resultado de la comparación de sus entradas, el cual entraría al siguiente nivel que buscaría el mínimo entre ellos. Por cada nivel, también se pasa el indice correspondiente, tal y como se define en la **Sección 3.5.2**. La configuración de puertos de este componente es el siguiente:

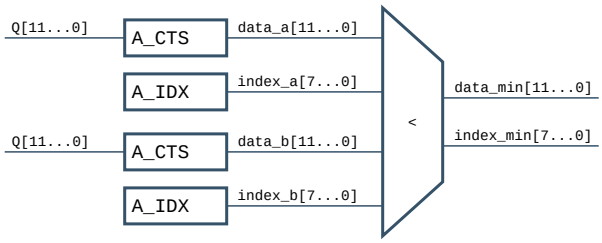


Figura 3.11: Bloque comparador.

- ***cost(64)[11...0]***. Se recibe un *buffer* que contiene todos los costes *SAD* de las salidas de cada uno de los procesadores. Cada coste tiene una profundidad de 12 bits, y el tamaño del *buffer* no debe de ser mayor a 64 (las disparidades establecidas).
- ***level[7...0]***. Puerto por donde se trasmite el valor mínimo encontrado de 8 bits.

Los datos transmitidos son almacenados en registros dentro de este componente. Para construir el árbol, es necesario usar la palabra reservada *generate* del VHDL para crear múltiples comparadores; el primero nivel va a tener 32 comparadores, el segundo la mitad de estos, es decir 16, el siguiente nivel la mitad hasta llegar a solo dos comparadores. El total de comparadores que tiene el árbol son de 63 componentes. El primer nivel de comparadores se asigna directamente los datos almacenados en los registros así como también se le asigna una posición, para el siguiente nivel se transmite los resultados de la comparación del nivel anterior así como su posición. Este proceso se hace por cada nivel hasta llegar al último nivel en el que solo se obtiene un único resultado, sin embargo, solo se toma su posición como el nuevo valor de el píxel.

Componente *comparator.vhd*

El árbol definido anteriormente implementa comparadores como cada nodo del árbol para hacer las comparaciones de los registros de 16 bits. Este componente fue diseñado siguiendo la lógica de un multiplexor, es decir, solo deja pasar el aquel valor menor y su posición para el siguiente nivel del árbol. Para este componente, se definen los siguientes puertos:

- ***data_a[11...0]***. Entrada de 12 bits donde se transmite el primer dato que se va a comparar.

- ***data_b[11...0]***. Entrada de 12 bits por donde se transmite el segundo dato que se va a comparar.
- ***index_a[7...0]***. Entrada de 8 bits donde se transmite la posición de el primer dato.
- ***index_b[7...0]***. Entrada de 8 bits donde se transmite la posición de el segundo dato.
- ***data_min[11...0]***. Puerto de salida de 12 bits por donde se transmite el resultado de la comparación, es decir, el mínimo entre los dos.
- ***index_min[7...0]***. Puerto de salida de 8 bits por donde se transmite la posición de el valor mínimo resultante de la comparación.

Las comparaciones se hacen implementando los operadores por defecto incluidos dentro de la librería *ieee.numeric_std*. Dentro de cada componente incluye un proceso que define los datos de entrada como señales de sensibilidad. Las comparaciones solo se hacen con los datos; si *data_a* es menor que *data_b*, entonces se asigna a la salida *data_min* el valor de *data_a*, en caso contrario se asigna el valor de *data_b*. Como también se requiere su posición, entonces se asigna a *index_min* el valor de *index_a* en caso de que el menor sea *data_a*, en caso contrario se asigna *index_b*. Cada nivel requiere toda la información para hacer las comparaciones, así que por cada nivel se transfiere los dato y su indice.

3.6. Diseño del microcontrolador

El FPGA es el que tiene la función de realizar el proceso principal de encontrar la correspondencia entre el par de imágenes. Sin embargo, es necesario llevar un control de estos procesos, siendo este el objetivo asignado al microcontrolador. Esta parte tiene la función de preparar los datos que se van a procesar, además de llevar el control de los procesos que se van a realizar dentro del FPGA. A diferencia de este último, la programación no se hace por medio de un lenguaje como VHDL si no que es posible emplear el lenguaje C con Vitis IDE de Xilinx. Este programa ya incluye las librerías para trabajar con FPGA, solo es necesario crear el archivo *bitstream* desde el Vivado para crear la plataforma en el programa Vitis.

El diseño de este microcontrolador es hecho por medio de dos archivos: *imeg.c* e *imeg.h*.

El primero contiene el código que se va a ejecutar una vez que se compile el programa, mientras que el archivo *imeg.h* contiene las cabeceras de las funciones prototipo y unas macros que permiten definir valores por defecto, ademas de incluir archivos adicionales para otras funciones.

3.6.1. Archivo de cabecera *imeg.h*

Dentro de esta cabecera es posible incluir macros y funciones que posibilitan la reutilización en el resto del código, ademas de incluir aquellos archivos que contiene funciones adicionales empleados a lo largo del programa. Tener el código ordenado de este modo facilita el hecho de que si es necesario cambiar el valor de una macro y incluir algún archivo más solo se tenga que hacer en esta sección sin necesidad de alterar el código principal. La estructura de este archivo se encuentra dividido en tres partes principales: la inclusión de los archivos de las cabeceras adicionales, las macros empleadas, y las funciones prototipo.

La inclusión de los archivos se hace mediante la directiva *#include* seguida de los signos *<file>*, en el interior se coloca la dirección del archivo a incluir en el proyecto. Los archivos que se están empleando en este proyecto son los siguientes:

- ***platform.h***. Contiene algunos atributos relacionados con el hardware que se está empleando, así como funciones que permite inicializar esta misma.
- ***xparameters.h***. Contiene los parámetros que definen las direcciones de los controladores de el hardware. Es aquí donde se puede obtener las direcciones de memoria para la lectura y escritura.
- ***ff.h***. Contiene funciones que permiten trabajar con el sistema de archivos. Para el empleo de esta librería, es necesario tener activado los puertos de la SD desde el FPGA.
- ***xil_types.h***. Contienen los tipos de datos optimizados para este procesador. Incluye la posibilidad de trabajar con números entero de cualquier tamaño, pero no incluye tipo de datos flotantes.
- ***xtime_l.h***. Contiene funciones que permiten hacer la estimación de tiempos.

- ***stdio.h***. Librería estándar de C que contiene las funciones y los atributos que permite trabajar con operaciones de entrada y salida. Es empleado en este proyecto para hacer la conversión de un valor de punto flotante a una cadena que pueda ser desplegada en la consola. La inclusión de este archivo provocaba un problema que y la solución a este se describe en la **Sección 3.5.1**.

Es muy importante que solo se incluyan aquellos archivos que solo van a ser empleados en el programa.

Las siguiente parte de la cabecera es la definición de las constantes. Estos valores se establecen mediante macros empleando la directiva *#define*, esto significa que donde coloquemos este macro, este va a ser reemplazado con la información establecida. Las primeras macros son definidas como funciones que permiten hacer la comparación entre dos números y obtener el máximo o mínimo (**Tabla 3.3**).

Nombre	Contenido	Descripción
MAX(a, b)	((a) > (b) ? (a) : (b))	Obtiene el mayor resultado de la comparación de entre dos números.
MIN(a, b)	((a) < (b) ? (a) : (b))	Obtiene el menor resultado de la comparación de entre dos números.

Tabla 3.3: Definición de las macros como función empleadas para la comparación de dos números y obtener el máximo o mínimo de entre los dos.

La siguiente parte contiene información relacionada con los atributos de las imágenes empleadas en el programa, como son sus dimensiones, el número total de píxeles, y el tamaño de la ventana empleado por el operador local (**Tabla 3.4**).

A cada una de las imágenes con la que se trabaja se le asigna una dirección de memoria por medio de una macro que luego es convertida en un puntero, indicando que cada imagen ya tiene su espacio en la memoria reservada. Tomando como referencia la dirección base $0x10000000_h$ se va sumando el tamaño de cada imagen dependiendo si esta tiene uno o tres canales. Las

Nombre	Contenido	Descripción
WIDTH	320	Número total de columnas de la imagen.
HEIGHT	240	Número total de filas de la imagen.
PARTS	2	Número de partes en la que se divide la imagen.
LENGHTx1C	$WIDTH \times HEIGHT$	Número de elementos que contiene el arreglo de píxeles de las imágenes de un solo canal.
LENGHTx3C	$WIDTH \times HEIGHT \times 3$	Número de elementos que contiene el arreglo de píxeles de las imágenes de tres canales.
WINDOWS_S	3	Tamaño de la ventana usado para la aplicación de el operador local.

Tabla 3.4: Definición de las macros que contienen información acerca de las propiedades de la imagen.

imágenes declaradas y sus direcciones se muestran en la siguiente tabla (**Tabla 3.5**).

Nombre	Contenido	Descripción
IMG_OR_LEFT	0x10000000	Imagen original izquierda.
IMG_OR_RIGHT	0x10038400	Imagen original derecha.
IMG_GR_LEFT	0x10070800	Imagen izquierda en escala de grises.
IMG_GR_RIGHT	0x10083400	Imagen derecha en escala de grises.
IMG_DEPH	0x10096000	Mapa de profundidades.
IMG_MEDIAN	0x100A8C00	Mediana del mapa de profundidades.

Tabla 3.5: Definición de las macros que contienen las direcciones de memoria asignadas a cada imagen.

A cada uno de los registros de 32 bits definidos en el controlador AXI se le asigna una dirección por medio de una macro, la cual es empleada para hacer la transferencia de datos. En la tabla siguiente se puede observar los 6 registros de 32 bits que fueron diseñados para este proyecto (**Tabla 3.6**).

Nombre	Contenido	Descripción
REG_SLV_0	0x00	Dirección del registro 0 del controlador AXI.
REG_SLV_1	0x01	Dirección del registro 1 del controlador AXI
REG_SLV_2	0x02	Dirección del registro 2 del controlador AXI
REG_SLV_3	0x03	Dirección del registro 3 del controlador AXI
REG_SLV_4	0x04	Dirección del registro 4 del controlador AXI
REG_SLV_5	0x05	Dirección del registro 5 del controlador AXI

Tabla 3.6: Definición de las macros que contienen las direcciones de los registros incluidos en el controlador AXI.

Por último se especifican los diferentes valores que puede tomar el registro 0 del controlador AXI indicando cada una de las opciones. El registro es de 32 bits y la información es descrita en hexadecimal indicando aquellos bits activados para activar determinada función. Esta información es descrita en la siguiente tabla (**Tabla 3.7**).

Nombre	Contenido	Descripción
VOID	0x00000000	Valor que toma el registro 0.
REINICIO	0x00000001	Valor que toma el registro 0 y esto hace que los contadores de el píxel se reinicien.
INICIO	0x00000002	Valor que toma el registro 0 y esto inicia el procesamiento.
ESCRITURA	0x00000004	Valor que toma el registro 0 y esto activa la escritura de la información en los bloques de memoria.
LECTURA	0x00000008	Valor que toma el registro 0 y esto activa la lectura de los píxeles de los bloques de memoria.

Tabla 3.7: Definición de las macros que contienen los valores que toma el registro 0 y que controla cada una de las fases de el procesamiento.

Al final del archivo cabecera se definen las funciones prototipo. Estas funciones son descritas

más adelante cuando se ve la implementación del código.

3.6.2. Archivo de código fuente *imeg.c*

Es en esta parte en donde se coloca el código del programa empleando las macros y funciones definidas anteriormente. En este archivo no es necesario la inclusión de más archivos salvo la cabecera creada anteriormente, que incluye las demás. Seguida de la inicialización de variables globales que se van a usar a lo largo del programa y que se describen a continuación.

La dirección base a donde se apunta los registro está almacenada en una macro definida por Xilinx, esta dirección es creada por defecto al momento de la creación del hardware y se encuentra en la cabecera *xparameters.h*. Esta dirección es almacenada en un entero de 32 bits para su utilización más adelante, y su declaración se puede ver a continuación.

```
u32 *baseaddr_p = (u32 *)XPAR_STEREO_MATCH_0_S00_AXI_BASEADDR;
```

Cabe mencionar de que el nombre de la macro depende del nombre que se haya asignado al la IP durante su creación.

Las siguientes variables que se definen son punteros a una dirección de memoria en donde se almacena cada una de las imágenes descritas en la **Tabla 3.5**. Cada uno de los punteros son enteros positivos de 8 bits, y van a tener diferente tamaño dependiendo si la imagen apuntada es de 24 u 8 bits. La definición se muestra a continuación:

```
u8 *img_or_left;
u8 *img_or_right;
u8 *img_gr_left;
u8 *img_gr_right;
u8 *img_deph;
u8 *img_median;
```

Los nombres de los archivos que contienen las imágenes originales se definen como arreglos de tipo carácter. Como se describe en la **Sección 3.4.1** los archivos son almacenado como un *stream* de bytes en un archivo de texto de extensión *.bit*. Los nombres de los archivos para este proyecto se muestran a continuación:

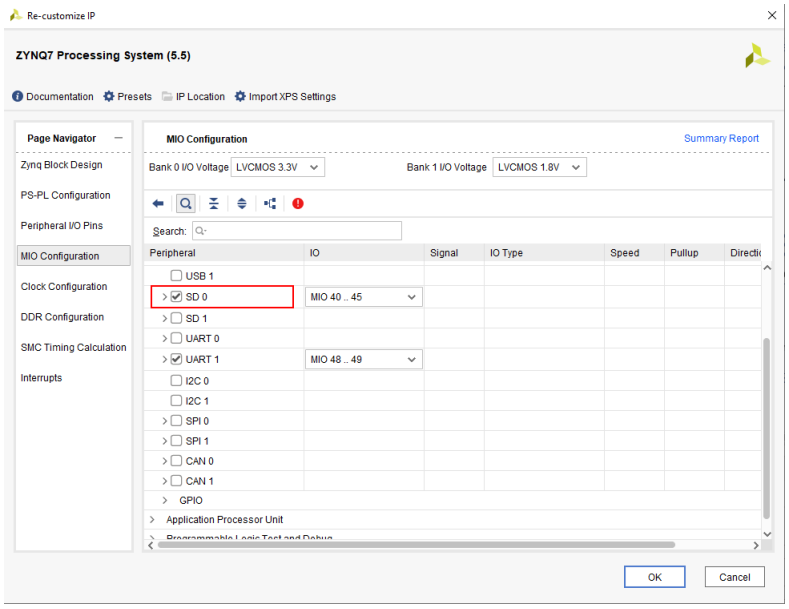


Figura 3.12: Activación del puerto SD desde el FPGA. Este se hace configurando el procesado *ZYNQ7 Processing System* colocado en el diagrama de bloques del Vivado.

```
static char load_img_left[7] = "lf.bit";
static char load_img_right[7] = "rg.bit";
static char save_deph[7] = "dp.bit";
```

Los archivos especificados anteriormente están almacenados en una memoria SD, sin embargo, para leerlos es necesario hacer el montaje de la unidad SD. El dispositivo FPGA tiene un controlador que permite la lectura de la SD sin necesidad de hacer intervención, y por medio de la inclusión del archivo ***ff.h*** es posible emplear las variables y funciones para el manejo de archivos. La declaración de los punteros de archivos se hace de la siguiente manera:

```
static FATFS fatfs;
static FIL file;
```

mientras que la función que hace el montaje y desmontaje de la unidad se muestra a continuación:

```
f_mount(&fatfs, path, 0);
```

Por medio de la función *f_mount()* contenida en el archivo ***ff.h*** permite hacer el montaje de la unidad, recibiendo como parámetros el puntero de la unidad y la dirección de la unidad que va a ser montada. El montaje de la unidad 0 se hace asignando a *path* el valor ”0:/”, mientras que para hacer el desmontaje se hace con la macro *NULL* en vez del puntero, y la misma dirección

de la unidad.

La carga de los archivos se hace empleando la variable *file* declarada anteriormente y una función por defecto contenido en el archivo ***ff.h***. El código para hacer la carga y el salvado de un archivo se muestra a continuación:

```
u8 load(u8 *_src, const char *_filename, u32 _lenght) {
    FRESULT result;
    UINT num_bytes_read;
    for(int i = 0; i < LENGHTx1C; i++) {
        *(_src + i) = 0x00;
    }
    result = f_open(&file, (char *)_filename, FA_READ);
    if (result != FR_OK) {
        return 0;
    }
    result = f_lseek(&file, 0);
    if(result != FR_OK) {
        return 0;
    }
    result = f_read(&file, (void*)_src, _lenght, &num_bytes_read);
    if(result != FR_OK) {
        return 0;
    }
    result = f_close(&file);
    if(result != FR_OK) {
        return 0;
    }

    return 1;
}
```

La función *load()* recibe como parámetros tres elementos: el puntero donde se va a almacenar el arreglo de píxeles, el nombre del archivo y el tamaño de este arreglo. Seguido se inicializan

todos los elementos del arreglo en ceros por medio de un ciclo *for*. A continuación se abre el archivo empleando la función *open()* contenida también en el archivo ***ff.h*** pasando el puntero de archivo declarado anteriormente, el nombre del archivo a leer y el parámetro *FA_READ* que indica que solo se está leyendo el archivo. La siguiente función *f_lseek()* es para posicionar el apuntador a la dirección cero del archivo, con esto permite hacer una lectura completa del archivo. Para leer la información se hace con la función *f_read()* pasando como parámetros el apuntador del archivo, el apuntador del arreglo donde se va a guardar la información y el número de bytes que se van a leer, para hacer la escritura del arreglo a un archivo solo se cambia la macro *FA_READ* por *FA_WRITE* en la función *open()* y se reemplaza la función *f_read()* por *f_write()* pasando los mismos parámetros. Al final de todo este proceso y como en los demás lenguajes, el archivo de tiene que cerrar con la función *f_close()*.

Las siguientes funciones son empleadas para el procesamiento de las imágenes que son leídas y enviadas al FPGA.

Las imágenes leídas pueden estar en escala de grises o a color. En caso de que estas estén a color, se puede hacer la conversión a escala de grises empleando la función *grayscale()* recibiendo como parámetros el puntero de la imagen a color original y el puntero de la imagen en escala de grises.

```
1 void grayscale(u8 *_src, u8 *_dst) {
2     u32 i = 0;
3     u32 j = 0;
4     for(i = 0; i < LENGHTx3C; i += 3) {
5         _dst[j] = (_src[i] + _src[i + 1] * 4 + _src[i + 2] * 2) / 7;
6         j += 1;
7     }
8 }
```

La función consiste en un ciclo *for* que recorre la imagen original que debe de ser a color, y va incrementando su indice en tres para que la nueva posición coincida con la nueva imagen en escala de grises. Por cada ciclo, se toman los tres elementos comenzando desde i hasta $i + 2$, y estos se operan empleando la **Ecuación 2.4** vista en secciones anteriores. El resultado de esta

se asigna a la nueva imagen.

La función de mediana, que aplica un filtro de suavizado de las regiones a la imagen es un *operador local*, esto quiere decir que se requieren los píxeles vecinos al píxel $I(x, y)$. La lectura de estos píxeles se hace por medio de una función que recibe el par de coordenadas, y retorna una dirección o posición al arreglo de la imagen. Como las imágenes están en disposición de un arreglo unidimensional, es necesario hacer esta operación para obtener el píxel de esta coordenada. La operación $addr = (y \times width) + x$ descrita en la **Sección 3.5.2** es la misma que se emplea en este caso y el código de su implementación se muestra a continuación:

```
void set_value(u8 *_src, u32 _x, u32 _y) {
    u32 x = _x;
    u32 y = _y * WIDTH;
    return _src[x + y];
}
```

Esta función asume que la imagen recibida está en escala de grises. Permite hacer la lectura de un byte del arreglo ubicado en una dirección determinada. Recibe como parámetros 3 elementos: el arreglo que contiene los píxeles y las coordenadas x y y de 32 bits. Para hacer la escritura en el arreglo es necesario también pasar como parámetro el byte que se va a escribir y a diferencia de este, no retorna un valor como resultado.

La función de mediana, simplemente llamada *median()* recibe como parámetros solo la imagen original de 8 bytes (escala de grises) y una arreglo nuevo en donde se almacena el resultado, también de 8 bytes. La estructura de esta función es la misma vista en el algoritmo presentado en la **Sección 2.4.2**, la diferencia se encuentra donde se crea la ventana ya que en este no aplica una máscara o *kernel* si no más bien aplica un algoritmo de ordenación a los nueve elementos de la ventana, y selecciona el elemento central de este. El código se visualiza a continuación:

```
void median(u8 *_src, u8 *_dst) {
    u8 W = WINDOWS_S;
    u8 FULL_SIZE = W * W;
    u8 MEDIANA = FULL_SIZE / 2;
    u8 data[FULL_SIZE];
```

```
    u8 cont = 0;
    for(u32 y = 0; y < HEIGHT - W - 2; y++) {
        for(u32 x = 0; x < WIDTH - W - 2; x++) {
            cont = 0;
            for(u8 j = 0; j < W; j++) {
                for(u8 i = 0; i < W; i++) {
                    data[cont] = get_value(_src, x + i, y + j);
                    cont++;
                }
            }
            quick_sort(data, FULL_SIZE);
            set_value(_dst, x, y, data[MEDIANA]);
        }
    }
}
```

Las dimensiones de la ventana ya fueron especificados en la **Sección 3.6.1**. Los recorridos de la imagen se hacen ignorando los bordes de esta.

El algoritmo de ordenación empleado para este proyecto es *Quicksort*. Esta función debe de ordenar los elementos de la ventana que se generado dentro de la función *median()* de meno a mayor, para tomar el elemento que se encuentra posicionado a la mitad del arreglo. El código de esta función está basado en el algoritmo mostrado en la **Sección 2.5** y se muestra a continuación:

```
void quick(u8 _A[], u8 _p, u8 _r) {
    if (_p >= _r) {
        return;
    }
    u8 q = partition(_A, _p, _r);
    quick(_A, _p, q - 1);
    quick(_A, q + 1, _r);
}

```

```
10 void quick_sort(u8 _A[], u8 _n) {
11     quick(_A, 1, _n - 1);
12 }
```

La función *quick_sort()* es la que se llama cuando va a ser empleada e inicializa los elemento para hacer las particiones, mientras que la función *quick()* es la que realiza las particiones hasta que ya no queda nada por seccionar. La función que hace el proceso de ordenación es la *partition()* y es la que se muestra a continuación:

```
1  u8 partition(u8 _A[], u8 _p, u8 _r) {
2      u8 x = _A[_r];
3      u8 i = _p - 1;
4      for(u8 j = _p; j < _r; j++) {
5          if(_A[j] <= x) {
6              i += 1;
7              swap(&_A[i], &_A[j]);
8          }
9      }
10     swap(&_A[i + 1], &_A[_r]);
11     return i + 1;
12 }
```

En esta última función, se está empleando la función *swap()*. Esta función permite hacer el intercambio de datos entre dos variables, sin embargo, como se están empleando arreglos, es necesario que se intercambien las direcciones para que el arreglo no se desordene. La función diseñada para este caso se muestra a continuación:

```
1 void swap(u8 *_a, u8 *_b) {
2     u8 c = *_a;
3     *_a = *_b;
4     *_b = c;
5 }
```

Finalmente, las funciones que se requieren para la comunicación con el FPGA son llamadas *get_ip_value* y *set_ip_value*. Estas funciones hacen la comunicación para la transferencia de datos a través de los registros definidos en la IP llamada *stereo_match* vista en la **Sección**

3.5.2.

```
u32 get_ip_value(u8 _register) {
    return *(baseaddr_p + _register);
}

void set_ip_value(u8 _register, u32 _value) {
    *(baseaddr_p + _register) = _value;
}
```

La función *get_ip_value* recibe la dirección de un registro para hacer la lectura de la información. El registro tiene que estar configurado en modo lectura desde las creación del controlador AXI, las direcciones de los registro se especificaron en las macros vistas anteriormente, mientras que la dirección del controlador es especificado durante la creación del hardware. La función *set_ip_value* permite hacer la escritura de un dato en un registro. A diferencia del anterior, en este también se recibe como parámetro la información que va a ser escrita. Como los registros son de 32 bits, entonces el tamaño de esta variable también es de 32 bits.

3.6.3. Función *main()*

Esta es la función principal del programa, ya que esta es llamada primero cuando se hace la compilación del código fuente. Así que todas las demás funciones definidas anteriormente son llamadas dentro de esta función para hacer el funcionamiento del programa. En la **Figura 3.13** se muestra un diagrama de flujo que representa los procesos de esta función.

La función está dividida principalmente en tres partes principales. La primera parte hace la inicialización de los arreglos que van a a contener cada uno de los vectores de los píxeles correspondientes a cada imagen, asignando la dirección de memoria correspondiente definidas anteriormente en las macros, además de hacer el montaje de la unidad, hacer la lectura de cada uno de los archivos y convertirlas a escala de grises. Posteriormente entra a un ciclo *while* que va a aplicar los siguientes procesos a cada una de las partes de las imágenes. En la **Sección 3.5.1** se abordaba el problema de porque no se puede transferir la imagen completa, entonces esta tuvo que ser segmentada en dos partes iguales de 38400 píxeles. Se hace la transferencia de una primera parte a los bloques de memoria del FPGA por medio de la dirección contenida en

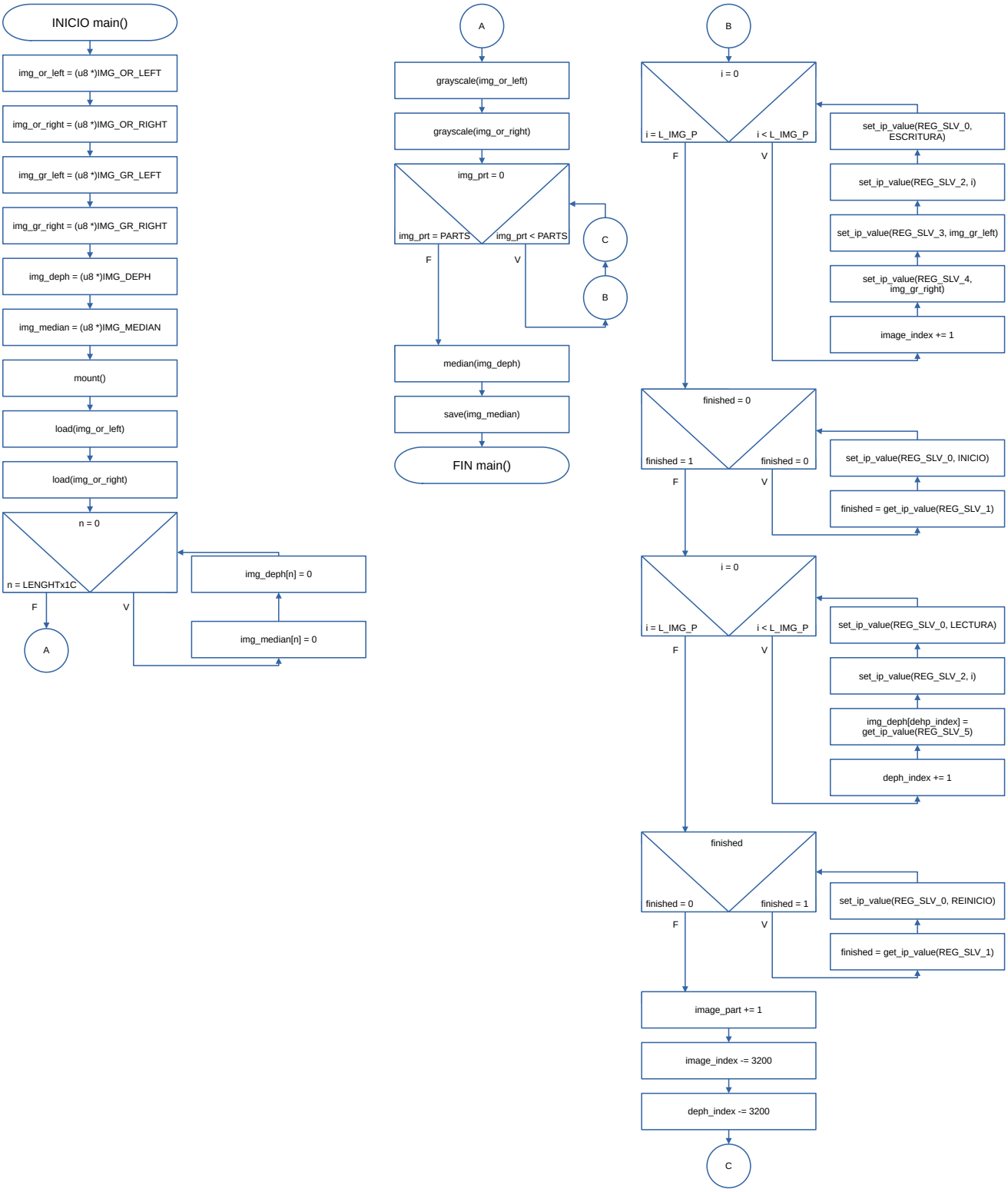


Figura 3.13: Diagrama de flujo de la función *main()*.

image_index, se emplea una variable adicional debido a que se tiene que dejar un margen extra de $width \times w$ siendo w el tamaño de la ventana. Este margen adicional es para que sea posible

combinar ambas partes una vez terminado el proceso. Cada uno de los píxeles es enviado al FPGA mediante la función *set_ip_value()* en el que se envía la dirección del registro donde va a ser escrito, pero antes también se tiene que activar la escritura con la macro *ESCRITURA* y transferir la dirección de escritura contenida en la variable *image_index*. Después de esto entra a un ciclo *while* en el que tiene que esperar a que la primera parte esté, leyendo constantemente el contenido del registro *REG_SLV_1* hasta que finaliza. Después hace la lectura de la información resultante empleando un ciclo *for* similar a el que hace la trasferencia, salvo que este lee otro registro y deposita la información en un nuevo arreglo e incrementa la variable *deph_index*. Finalmente reinicia los registros y espera a que se coloque en inicio mediante un ciclo*while*. Al final, a las variables *image_index* y *deph_index* se les resta 3200 cambiar la posición del puntero y lograr la combinación de las partes de las imágenes. Por último, en la última sección de la función *main()* solo se aplica un filtro de mediana a la imagen resultante para reducir el número de partículas. La imagen resultante de este filtro es almacenada en la SD en un nuevo archivo.

Capítulo 4

Resultados

En la **Figura 4.1** se muestra el diseño completo del *hardware*; en el se definen el procesador principal con nombre *ZYNQ7 PROCESSING SYSTEM* y algunos componentes extras que trabajan en conjunto con este, también se visualiza el componente creado para este proyecto con nombre *stereo_match* en el que incluye el código visto en las secciones anteriores.

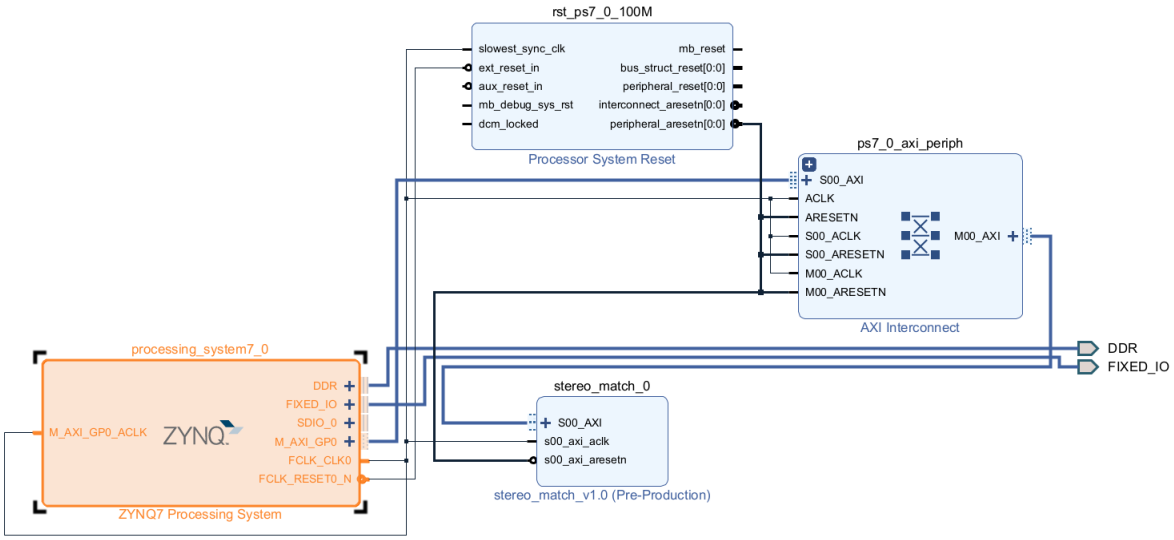
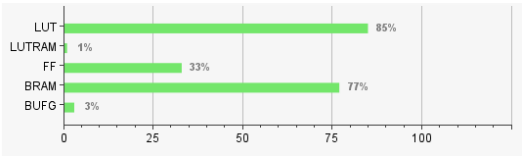


Figura 4.1: Diagrama de bloques del proyecto capturado del programa Vivado.

El consumo de recursos del FPGA es debido a que el algoritmo planteado consume recursos para almacenar valores temporales y reutilizar cierta información acelerando los tiempos de procesamiento, sin embargo, todavía queda algunas cosas que se pueden reducir ya que se les dejó algunos componentes con una lógica secuencial que generan un retraso a propósito para

asegurar los resultados finales. El consumo de recursos se puede visualizar en la **Figura 4.2**, mientras que el consumo energético se puede visualizar en la **Figura 4.3**.



(a) Consumo de recursos en porcentaje.

Resource	Utilization	Available	Utilization %
LUT	15006	17600	85.26
LUTRAM	60	6000	1.00
FF	11466	35200	32.57
BRAM	46	60	76.67
BUFG	1	32	3.13

(b) Consumo de recursos por unidad.

Figura 4.2: En esta figura se muestra el consumo total de recursos en el proyecto. Los resultados fueron obtenidos en el programa de Vivado.

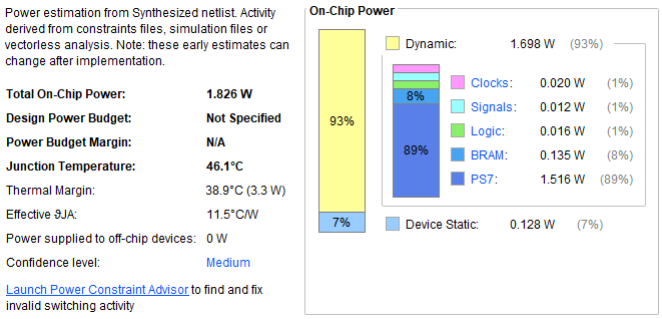


Figura 4.3: Consumo energético del proyecto.

Los resultados del procesamiento se obtuvieron mediante el software de Vitis, por medio de la librería *xtime_l.h* se hizo un conteo de los ciclos de reloj y una estimación de tiempo en milisegundos desplegando la información en la misma consola de los resultados parciales y totales de cada una de las secciones del programa. En la siguiente **Tabla 4.1** se recogen los resultados del tiempo del procesamiento de cada una de las partes de la función *main()*.

En la parte que consume más tiempo es aplicar el filtro de mediana a la imagen resultante. Esta no es una parte esencial del programa si no lo que importa aquí es el tiempo de procesamiento en hacer la correlación del par de imágenes. Tan solo toma aproximadamente 160 ms a diferencia de aplicar un formato secuencial en un lenguaje de alto nivel que esto se puede llegar a incrementar hasta 5 veces (implementar el mismo algoritmo en el microcontrolador tomó alrededor de 900 ms), sin embargo, a pesar de que es mucho más rápido que su implementación

Función	Ciclos de reloj	Tiempo (en milisegundos)
Escala de grises	$\sim 6,977,896$	$\sim 10.7352\,ms$
Transferencia de imágenes	$\sim 46,868,220$	$\sim 72.105\,ms$
Búsqueda de correspondencia	$\sim \mathbf{103,849,632}$	$\sim \mathbf{159.7686\ ms}$
Recuperación de datos	$\sim 32,236,024$	$\sim 49.5942\,ms$
Reinicio	~ 326	$\sim 0.0005\,ms$
Mediana	$\sim 168,930,362$	$\sim 259.89\,ms$
Total	$\sim 358,862,460$	$\sim 552.0935\,ms$

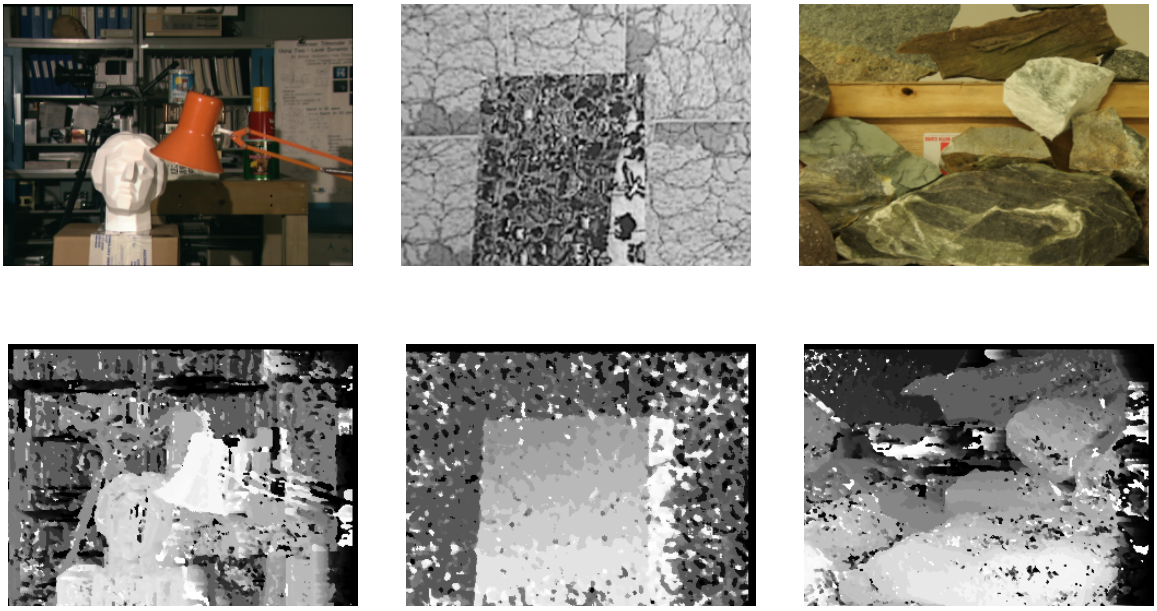
Tabla 4.1: Tiempos de procesamiento aproximados que toma cada una de las partes de la función *main()*. El proceso remarcado corresponde al FPGA.

en software, todavía se puede hacer que esto vaya más rápido, esto es debido a que a lo largo del programa se menciona de que se colocaron algunos procesos que producen algún retraso, esto se dejó a propósito para asegurar los resultados. Cambiando de lógica en esta sección del diseño es posible que vaya aún más rápido.

Sin embargo, en las **Figuras 4.4 y 4.5** se puede observa que en las imágenes resultantes de la implementación hay muchas partículas, también conocidos como *false matches*. Estas partículas son generadas debido a que la ventana del operador local es demasiado pequeña (en la **Sección 3.5.1** se menciona que debido a los recursos del FPGA son limitado, el tamaño de la ventana tuvo que ser reducido). Empleando los *ground truth* y una función de la librería OpenCV se puede calcular el error cuadrático para obtener la cantidad de *false matches*. Los resultados del cálculo aplicados a las imágenes de la **Figura 4.4** son los siguientes:

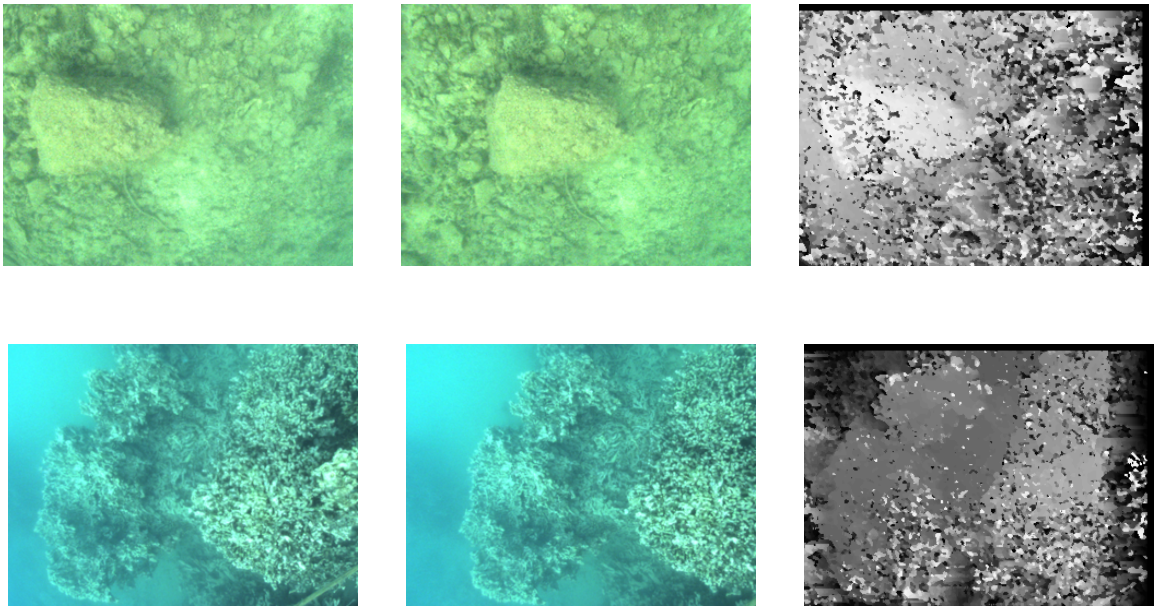
- *Tsukuba*: 73 %.
- *Rocks*: 70 %.
- *Map*: 57 %.

Debido a que las imágenes resultantes tienen muchas partículas causadas por los *false matches* los resultados siguen estando cerca a lo esperado. En la literatura se menciona que el porcentaje obtenido por medio de hardware están entre el 80 % y 90 % [7].



(a) Tsukuba (b) Map (c) Rocks

Figura 4.4: Resultados de la implementación del algoritmo en el hardware. Las imágenes ya rectificas fueron obtenidas de la base de datos de *Middlebury* en el sitio `https://vision.middlebury.edu/stereo/data/` [23] [27]. Las imágenes pasaron por un filtro de ecualización de histograma externo para que se pudieran visualizar.



(a) Izquierda (b) Derecha (c) Mapa de profundidad

Figura 4.5: Resultados de la implementación del algoritmo en el hardware. Las imágenes son de una base de datos de entrenamiento de redes neuronales para la generación de mapas de profundidad, recuperadas del sitio `https://github.com/kskin/data` [41]. Las imágenes pasaron por un filtro de ecualización de histograma para que se pudieran visualizar.

Capítulo 5

Conclusiones

La estimación de la profundidad es un proceso demandante en el que ha habido diferentes maneras de resolver el problema tanto en software como en hardware según la literatura revisada. Sin embargo, su implementación en un dispositivo limitado como un FPGA ha supuesto un desafío debido a que se ha tenido que renunciar a la calidad para ganar velocidad en la obtención de los resultados, sobre todo para que pueda ser implementado en un sistema de navegación como se propone. Además, en la navegación submarina es necesario tomar a consideración algunos procesos para hacer correctamente la captura y el filtrado de las imágenes de acuerdo a las características del medio, ya que debido a la atenuación de la luz en el medio acuático ha provocado que el reconocimiento de los objeto cause falsos *matches* en los mapas de disparidad obtenidos.

5.1. Trabajo a futuro

Como se mencionaba anteriormente, en algunas partes del programa se colocaron algunos elementos auxiliares como *flip-flops* para asegurar de los resultados se están escribiendo correctamente, pero esto genera algún retraso en el procesamiento. El foco de interés para un futuro lector sería retirar estos elementos a favor de aumentar la velocidad de procesamiento. También, durante la lectura de los *buffers* que contiene el par de imágenes en los bloques de memoria, se están realizando múltiples lecturas para escribirlas en los *buffers* temporales de cada procesador (ver **Sección 3.5.2**), en esta parte se puede plantear la lógica de modo que, cada procesador tenga un *buffer* compartido que sería la última columna de la ventana izquierda, para que el

procesador anterior haga lectura de esta columna y la escriba como nueva información, haciendo este proceso por cada uno de los 64 procesadores. Esto haría que en la primera iteración, cuando $x = 0$, entonces se crearían las ventanas haciendo la lectura completa de 64×3, y por cada nuevo valor de x tan solo se leería la columna nueva, reduciendo las lecturas de 64×3 a tan solo 3 lecturas por cada valor de x. En la **Figura 5.1** queda ejemplificado esta propuesta de diseño.

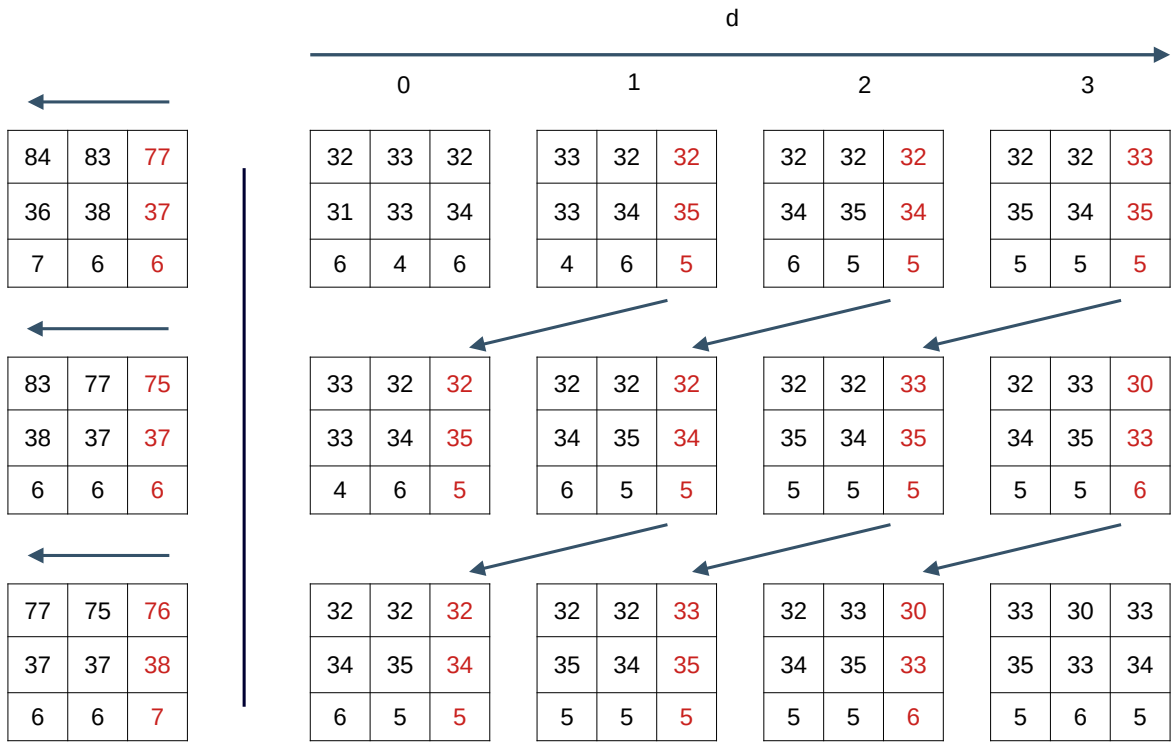


Figura 5.1: Propuesta para el intercambio de información entre los procesadores.

Otro trabajo pendiente es diseñar un controlador especifico para las cámaras, pero esta vez tomando en cuenta también algoritmos de filtrado para el medio en el que se va a emplear el sistema. Esto es necesario si se quiere implementar en un sistema de navegación autónoma.

5.2. Repositorio

En las siguientes carpetas está almacenado el código del proyecto, tanto en C, C++ y en VHDL. Las carpetas están divididas en dos: *bmp* contiene el código para la carga y guardado de imágenes con extensión *.bmp* mientras que *zybo* contiene todo el código del diseño del *hardware* en VHDL. El hipervínculo para la carpeta en Github es `https://github.com/`

`mcastr0m/zybo-z7` mientras que para Google Drive es `https://drive.google.com/drive/folders/1NJuW3S7b3jRKQa2TxbSQTS4EM8gtJ-ME`.

Apéndice A

Código fuente para la creación de las cabeceras de archivo en C++

A.1. Cabeceras de archivo de imagen *.bmp*

```
#pragma pack(push, 1)
struct img_bmp_file_header {
    uint16_t  FILE_TYPE     {0x4D42};
    uint32_t  FILE_SIZE     {0};
    uint32_t  RESERVED      {0};
    uint32_t  OFFSET_DATA   {0x00000036};
};

struct img_bmp_info_header {
    uint32_t HEADER_SIZE    {0x00000028};
    uint32_t IMG_WIDTH      {0};
    uint32_t IMG_HEIGHT     {0};
    uint16_t IMG_PLANES     {0x0001};
    uint16_t BITS_PER_PIXEL {0};
    uint32_t COMPRESSION    {0};
    uint32_t SIZE_IMAGE     {0};
    int32_t  X_RESOLUTION   {0x00000EC4};
```

```
18      int32_t  Y_RESOLUTION   {0x00000EC4};
19      uint32_t COLORS_USED    {0};
20      uint32_t COLORS_IMPT    {0};
21  };
22
23  struct img_bmp_color_header {
24      uint32_t RED            {0x00FF0000};
25      uint32_t GREEN          {0x0000FF00};
26      uint32_t BLUE           {0x000000FF};
27      uint32_t ALPHA          {0xFF000000};
28      uint32_t SPCOLOR_TYPE   {0x73524742};
29      uint32_t UNUSED[16]     {0};
30  };
31  #pragma pack(pop)
```

Apéndice B

Código fuente del algoritmo de búsqueda de la correlación en C++

B.1. match()

```
1  void match(IMAGE _left, IMAGE _right, IMAGE _deph, unsigned char _w = 7U,
       unsigned char _d = 64U) {
2      if(_left.channels() != 1 && _right.channels() != 1 && _deph.channels()
           != 1)
3          return;
4
5      const int DISPARITY = _d;
6      const int W = _w;
7      const int MW = W / 2;
8      double energia = 0.0f;
9      double min = 0.0f;
10     int absmin = 0;
11     double **LEFT_W = new double *[W];
12     double **RGHT_W = new double *[W];
13     for(int i = 0; i < W; i++) LEFT_W[i] = new double[W] { 0.0 };
14     for(int i = 0; i < W; i++) RGHT_W[i] = new double[W] { 0.0 };
15     double *BUFFER = new double[DISPARITY] { 0x00 };
```

```
16
17      for(int x = W; x < _left.width() - W; x++) {
18          for(int y = W; y < _left.height() - W; y++) {
19              absmin = 0;
20              min = 0;
21
22              for(int i = -MW; i <= MW; i++) {
23                  for(int j = -MW; j <= MW; j++) {
24                      LEFT_W[i + MW][j + MW] = (double)_left.at(x + i, y + j
                            , 0);
25                  }
26              }
27
28              for(int d = 0; d < DISPARITY; d++) {
29                  energia = 0.0;
30
31                  for(int i = -MW; i <= MW; i++) {
32                      for(int j = -MW; j <= MW; j++) {
33                          RGHT_W[i + MW][j + MW] = (double)_right.at(x + i +
                                d, y + j, 0);
34                      }
35                  }
36
37                  for(int i = 0; i < W; i++) {
38                      for(int j = 0; j < W; j++) {
39                          energia += abs(LEFT_W[i][j] - RGHT_W[i][j]);
40                      }
41                  }
42
43                  BUFFER[d] = energia;
44              }
45
46              min = BUFFER[0];
```

```
47              absmin = 0;
48              for(int i = 1; i < DISPARITY; i++){
49                  min = MIN(min, BUFFER[i]);
50                  absmin = min < BUFFER[i] ? absmin : i;
51              }
52
53              _deph.at(x, y, 0, absmin);
54          }
55      }
56
57      for(int i = 0; i < _deph.get_real_image_size(); i++)
58      _deph.first()[i] = ((float)_deph.first()[i] / (float)DISPARITY) * 256;
59
60      delete[] BUFFER;
61      delete[] LEFT_W;
62      delete[] RGHT_W;
63  }
```

Apéndice C

Código fuente para definir el archivo Make de la IP

C.1. *Makefile*

```
COMPILER=
ARCHIVER=
CP=cp
COMPILER_FLAGS=
EXTRA_COMPILER_FLAGS=
LIB=libxil.a

RELEASEDIR=../../../lib
INCLUDEDIR=../../../include
INCLUDES=-I./. -I${INCLUDEDIR}

INCLUDEFILES=*.h
LIBSOURCES=$(wildcard *.c)
OBJECTS = $(addsuffix .o, $(basename $(wildcard *.c)))
ASSEMBLY_OBJECTS = $(addsuffix .o, $(basename $(wildcard *.S)))

libs:
```

```
18      echo "Compiling [nombre de la ip]..."
19      $(COMPILER) $(COMPILER_FLAGS) $(EXTRA_COMPILER_FLAGS) $(INCLUDES) $(
            LIBSOURCES)
20      $(ARCHIVER) -r ${RELEASEDIR}/${LIB} ${OBJECTS} ${ASSEMBLY_OBJECTS}
21      make clean
22
23  include:
24      ${CP} $(INCLUDEFILES) $(INCLUDEDIR)
25
26  clean:
27      rm -rf ${OUTS}
```

Bibliografía

[1] Mario A. Castro-Morgan and Jose V. Flores-Ojeda. Detector de objetos mediante visión artificial, posición y orientación de las extremidades del sistema robótico humanoide poppy. Reporte de residencias profesionales, Tecnnológico Nacional de México / Instituto Tecnológico de La Paz, 2018.

[2] Prabhpreet Kaur, Gurvinder Singh, and Parminder Kaur. A review of denoising medical images using machine learning approaches. *Bentham Science Publishers*, 14:675 – 685, 2018.

[3] Sreedhar Kollem, Katta Ramalinga Reddy, and Duggirala Srinivasa Rao. A review of image denoising and segmentation methods based on medical images. *International Journal of Machine Learning and Computing*, 9:288 – 295, 2019.

[4] Seiichi Uchida. Image processing and recognition for biological images. *Develpment, Growth and Differentiation*, 55:523 – 549, 2013.

[5] Ya-Wen Hsu, Kai-Quan Zhong, Jau-Woei Perng, Tang-Kai Yin, and Chia-Yen Chen. Developing an on-road obstacle detection system using monovision. *2018 International Conference on Image and Vision Computing New Zealand (IVCNZ)*, pages 1 – 9, 2018.

[6] Masahiro Nakamura and Norishige Fukushima. Fast implementation of box filtering. *Proc. International Workshop on Advanced Image Technology (IWAIT)*, pages I – I, 2017.

[7] Luigi Di-Stefano, Massimiliano Marchionni, and Stefano Mattocia. A fast area-based stereo matching algorithm. *Image and Vision Computating*, 22:983 – 1005, 2004.

[8] Lifeng He, Yuyan Chao, Kenji Suzuki, and Kesheng Wu. Fast connected-component labeling. *Pattern Recognition*, 42:1977 – 1987, 2009.

[9] Chenxi Zhang, Brian Price, Scott Cohen, and Ruigang Yang. High-quality stereo video matching via user interaction and space-time propagation. *2013 International Conference on 3D Vision - 3DV 2013*, pages 71 – 78, 2013.

[10] A.S. Ogale and Y. Aloimonos. Robust contrast invariant stereo correspondence. *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 819 – 824, 2005.

[11] N. M. Sari, J. T. Nugroho, G. A. Chulafak, and D. Kushardono. Study of 3d bathymetry modelling using lapan surveillance unmanned aerial vehicle 02 (lsu-02) photo data with stereo photogrammetry technique, wawaran beach, pacitan, east java, indonesia. *IOP Conference Series: Earth and Environmental Science*, 149:1 – 1, 2018.

[12] Jiali Wang, Ming Chen, Weidong Zhu, Liting Hu, and Yasong Wang. A combined approach for retrieving bathymetry from aerial stereo rgb imagery. *Remote Sensing*, 14:1 – 1, 2022.

[13] F. Oleari, F. Kallasi, D. L. Rizzini, J. Aleotti, and S. Caselli. An underwater stereo vision system: From design to deployment and dataset acquisition. *OCEANS 2015 - Genova*, pages 1 – 6, 2015.

[14] Tomasz Łuczyński, Piotr Łuczyński, Lukas Pehle, Manfred Wirsum, and Andreas Birk. Model based design of a stereo vision system for intelligent deep-sea operations. *Measurement*, 144:298 – 310, 2019.

[15] Camilo Sánchez-Ferreira, Jones Y. Mori, Mylène C. Q. Farias, and Carlos H. Llanos. A real-time stereo vision system for distance measurement and underwater image restoration. *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, 38:2039 – 2049, 2016.

[16] D. Berman, D. Levy, S. Avidan, and T. Treibitz. Underwater single image color restoration using haze-lines and a new quantitative dataset. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43:2822 – 2837, 2021.

[17] D. Marr, T. Poggio, Ellen C. Hildreth, and W. Eric L. Grimson. A computational theory of human stereo vision. *From the Retina to the Neocortex*, pages 263 – 295, 1991.

[18] Seunghun Jin, Junguk Cho, Xuan Dai Pham, Kyoung Mu Lee, Sung-Kee Park, Munsang Kim, and Jae Wook Jeon. Fpga design and implementation of a real-time stereo vision system. *IEEE Transactions on Circuits and Systems for Video Technology*, 20:15 – 26, 2010.

[19] Jian Sun, Nan-Ning Zheng, and Heung-Yeung Shum. Stereo matching using belief propagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25:787 – 800, 2003.

[20] Nalpantidis Lazaros, Georgios Christou Sirakoulis, and Antonios Gasteratos. Review of stereo vision algorithms: from software to hardware. *International Journal of Optomechatronics*, 2:435 – 462, 2008.

[21] Tinne Tuytelaars and Luc Van Gool. Wide baseline stereo matching based on local, affinely invariant regions. *In Proc. BMVC*, pages–, 2000.

[22] Christian Banz, Sebastian Hesselbarth, Holger Flatt, Holger Blume, and Peter Pirsch. Real-time stereo vision system using semi-global matching disparity estimation: Architecture and fpga-implementation. *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 0 – 0, 2010.

[23] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, pages 7 – 42, 2002.

[24] Sukjune Yoon, Sung-Kee Park, Sungchul Kang, and Yoon Keun Kwak. Fast correlation-based stereo matching with the reduction of systematic errors. *Pattern Recognition Letters*, 26:2221 – 2231, 2005.

[25] Yosuke Miyajima and Tsutomu Maruyama. A real-time stereo vision system with fpga. *Field Programmable Logic and Application*, 2778:448 – 457, 2003.

[26] Elisabetta Binaghi, Ignazio Gallo, Giuseppe Marino, and Mario Raspanti. Neural adaptive stereo matching. *Pattern Recognition Letters*, 25:1743 – 1758, 2004.

[27] Heiko Hirschmüller and Daniel Scharstein. Evaluation of cost functions for stereo matching. *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1 – 8, 2007.

[28] James D. Murray and William vanRyper. *Encyclopedia of Graphics File Formats*. O'Reilly and Associates, Inc., 103 Morris Street, Suite A, Sebastopol, CA 95472, 2nd edition, 1996.

[29] Richard Szeliski. *Computer Vision*. Springer, Cham, Switzerland, 2nd edition, 2022.

[30] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. Massachusetts Institute Technology, Cambridge, Massachusetts, 3rd edition, 2009.

[31] Pong P. Chu. *FPGA Prototyping by VHDL Examples*. John Wiley & Sons, Hoboken, New Jersey, 2008.

[32] M. Morris Mano. *Diseño Digital*. Pearson Educación, México, 3ra edición edition, 2003.

[33] Xilinx. *Zynq-7000 SoC Technical Reference Manual*. Xilinx, 04 2021.

[34] Carlos Cuadrado, Aitzol Zuloaga, Jose L. Martin, Jesus Laizaro, and Jaime Jimenez. Real-time stereo vision processing system in a fpga. *IECON 2006 - 32nd Annual Conference on IEEE Industrial Electronics*, pages 3455 – 3460, 2006.

[35] Nilufar Isakova, Selçuk Başak, and A. Coşkun Sönmez. Fpga design and implementation of a real-time stereo vision system. *2012 International Symposium on Innovations in Intelligent Systems and Applications*, pages 1 – 5, 2012.

[36] D. K. Masrani and W. J. MacLean. A real-time large disparity range stereo-system using fpgas. *Fourth IEEE International Conference on Computer Vision Systems (ICVS'06)*, pages 13 – 13, 2006.

[37] Sungchan Park and Hong Jeong. Real-time stereo vision fpga chip with low error rate. *2007 International Conference on Multimedia and Ubiquitous Engineering (MUE'07)*, pages 751 – 756, 2007.

[38] Digilent. *Zybo Z7 Board Reference Manual*. Digilent, 02 2018.

[39] Gerard Medioni and Ramakant Nevatia. Segment-based stereo matching. *Computer Vision, Graphics, and Image Processing*, 31:2 – 18, 1985.

[40] Xilinx. *Zynq-7000 SoC Data Sheet: Overview*. Xilinx, 07 2018.

[41] Katherine A. Skinner, Junming Zhang, Elizabeth A. Olson, and Matthew Johnson-Roberson. Uwstereonet: Unsupervised learning for depth estimation and color correction of underwater stereo imagery. *2019 International Conference on Robotics and Automation (ICRA)*, pages 7947 – 7954, 2019.