ULPGC Universidad de Las Palmas de Gran Canaria | eite ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

DESARROLLO DE UNA PLATAFORMA DE PERCEPCIÓN ESTEREOSCÓPICA DE PROFUNDIDAD BASADA EN SOPC

Titulación:	Grado en Ingeniería en Tecnologías de la Telecomunicación
Mención:	Sistemas Electrónicos
Autor:	Santiago Díaz Villastrigo
Tutor:	Dr. Valentín de Armas Sosa
	Dr. Félix Bernardo Tobajas Guerrero
Fecha:	Julio 2025

AGRADECIMIENTOS

A mi familia por su apoyo incondicional.

A mis tutores, Valentín de Armas Sosa y Félix Bernardo Tobajas Guerrero, por su paciencia y orientación.

RESUMEN

En este Trabajo Fin de Grado se presenta el desarrollo de una plataforma *hardware/software* orientado a la implementación de un sistema que permita extraer información tridimensional de entornos reales mediante técnicas de visión artificial basadas en *estereopsis*. La visión estereoscópica es una técnica eficiente para la extracción de información de profundidad a partir del análisis de la disparidad entre imágenes capturadas desde diferentes posiciones.

La plataforma se basa en la placa de prototipado PYNQ-Z2 que integra un APSoC (*All Programmable System on Chip*) de la familia Zynq-7000 de AMD/Xilinx. Esta familia de dispositivos combina un procesador ARM y lógica programable FPGA en un solo chip, permitiendo una integración flexible de los componentes de la plataforma.

El proyecto se centra en desarrollar un sistema que incorpore un array de dos cámaras OV5640 conectadas mediante el protocolo SCCB, con el fin de generar el mapa de disparidad abordado con la técnica de emparejamiento local denominada *block matching*. El proceso completo implicará las etapas críticas de captura y alineación de imágenes en tiempo real sobre el dispositivo Zynq-7000, la implementación del algoritmo de emparejamiento estéreo local basado en la técnica de Block Matching (SLBM), la calibración y rectificación geométrica de las cámaras (realizada offline usando librerías como OpenCV), y finalmente la generación de un mapa de profundidad.

La solución propuesta contempla adicionalmente el diseño de una PCB específica para la conexión del array de cámaras con la placa de prototipado. Este componente, sumado a los núcleos IP personalizados y a la infraestructura *hardware/software*, permitirá la implementación de un prototipo que prioriza la flexibilidad de configuración. Además, se elaborará un conjunto de pruebas que evalúen la precisión de la reconstrucción estereoscópica y la robustez del sistema ante diferentes condiciones de iluminación o movimiento.

La implementación se llevará a cabo mediante herramientas específicas del entorno de AMD/Xilinx, incluyendo Vivado para la integración de los elementos de la arquitectura de la

plataforma y Vitis HLS para el desarrollo de bloques aceleradores *hardware*. La biblioteca Vitis Vision será clave para facilitar la implementación de algoritmos de procesamiento de imágenes descritos en lenguaje C/C++, simplificando así su conversión a hardware sintetizable sobre FPGA. Además, OpenCV se utilizará para los procesos de calibración y rectificación de imágenes, garantizando la calidad y alineación correcta previa al procesamiento en tiempo real.

El resultado esperado es una plataforma *hardware/software* funcional, capaz de capturar y procesar imágenes estereoscópicas de manera eficiente, con aplicaciones potenciales en diversos campos tecnológicos y con un análisis de rendimiento y precisión.

ABSTRACT

This Final Degree Project presents the development of a hardware/software platform aimed at implementing a system capable of extracting three-dimensional information from real environments through computer-vision techniques based on stereopsis. Stereoscopic vision is an efficient technique for obtaining depth information by analyzing the disparity between images captured from different positions.

The platform is built around the PYNQ-Z2 prototyping board, which integrates an APSoC (All-Programmable System-on-Chip) from the AMD/Xilinx Zynq-7000 family. These devices combine an ARM processor with FPGA programmable logic on a single chip, enabling flexible integration of the platform's components.

The project focuses on developing a system that incorporates an array of two OV5640 cameras connected via the SCCB protocol, in order to generate a disparity map using the local matching technique known as block matching. The complete process will include the critical stages of real-time image capture and alignment on the Zynq-7000 device, implementation of the stereo local matching algorithm based on Block Matching (SLBM), offline calibration and geometric rectification of the cameras with libraries such as OpenCV, and finally the generation of a depth map.

The proposed solution also involves designing a specific PCB to connect the camera array to the prototyping board. Together with custom IP cores and the hardware/software infrastructure, this will enable the implementation of a prototype that prioritizes configuration flexibility. A set of tests will be developed to evaluate the accuracy of the stereoscopic reconstruction and the robustness of the system under different lighting or motion conditions.

Implementation will use AMD/Xilinx tools, including Vivado for integrating the platform's architectural elements and Vitis HLS for developing hardware accelerator blocks. The Vitis Vision library will be essential for implementing image-processing algorithms described in C/C++, simplifying their conversion to synthesizable hardware on FPGA. OpenCV

will also be used for image calibration and rectification, ensuring quality and correct alignment prior to real-time processing.

The expected outcome is a functional hardware/software platform capable of efficiently capturing and processing stereoscopic images, with potential applications in various technological fields and accompanied by an analysis of performance and accuracy.

TABLA DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE CÓDIGOS

GLOSARIO

- **ACP** (Accelerator Coherency Port): Puerto de coherencia que conecta la lógica programable con la caché del procesador, ofreciendo baja latencia en el intercambio de datos.
- **ADC** (Analog-to-Digital Converter): Convertidor analógico-digital que discretiza señales analógicas en valores digitales.
- **APSoC** (All Programmable System-on-Chip): Sistema en chip totalmente programable que combina procesador y lógica reconfigurable.
- **ARM** (Advanced RISC Machines): Arquitectura de CPU RISC empleada en el subsistema de procesamiento (PS).
- **AXI** (Advanced eXtensible Interface): Protocolo de bus de alta velocidad definido por ARM para la interconexión de bloques en SoCs.
- **AXI4-Stream**: Variante del protocolo AXI orientada a flujos continuos de datos sin direcciones.
- **AXI-Lite**: Subconjunto ligero de AXI usado para acceso a registros de control de baja latencia.
- **AXI-HP / AXI-HPC**: Puertos AXI de Alto Rendimiento (High Performance / High Performance Coherent) para transferencias PS-PL de gran ancho de banda.
- **BRAM** (Block Random Access Memory): Memoria de acceso aleatorio integrada en la FPGA optimizada para altas velocidades.
- **BT.601**: Recomendación ITU-R que define la codificación Y'CbCr para vídeo de definición estándar.
- **BT.709**: Recomendación ITU-R que define la codificación Y'CbCr para vídeo de alta definición.
- **CMOS** (Complementary Metal-Oxide–Semiconductor): Tecnología de fabricación usada en sensores de imagen y circuitos integrados.
- **DDR** (Double Data Rate SDRAM): Memoria dinámica que transfiere datos en ambos flancos del reloj.
- **DMA** (Direct Memory Access): Mecanismo que permite a periféricos transferir datos a memoria sin intervención de la CPU.
- **DVI** (Digital Visual Interface): Interfaz digital de vídeo predecesora de HDMI.
- **DVP** (Digital Video Port): Interfaz paralela de salida de vídeo de 8/10 bits usada en sensores de cámara.
- **EMIO** (Extended Multiplexed I/O): Extensión que lleva señales adicionales del PS a la lógica programable en dispositivos Zynq.

- **FPGA** (Field-Programmable Gate Array): Matriz de puertas lógicas reconfigurable en campo por el usuario.
- **FT2232HQ**: Circuito integrado USB de FTDI que implementa puentes UART/FIFO de propósito general.
- **HDMI** (High-Definition Multimedia Interface): Interfaz digital audiovisual que transporta vídeo y audio.
- **HLS** (High-Level Synthesis): Proceso de generar hardware a partir de descripciones en C/C++.
- **HSYNC/HREF** (Horizontal SYNC/Reference): Señal que indica el inicio y la región activa de cada línea de vídeo.
- **HTS** (Horizontal Total Size): Número total de píxeles por línea, incluidos los periodos de porche horizontales.
- **I²C** (Inter-Integrated Circuit): Bus serie sincrónico de dos hilos para comunicaciones de baja velocidad.
- **IP** (Intellectual Property Core): Bloque funcional reutilizable dentro de un diseño FPGA.
- **ISP** (Image Signal Processor): Procesador integrado en la cámara que aplica corrección y conversión al flujo de imagen.
- **JTAG** (Joint Test Action Group): Interfaz estándar para prueba y programación de dispositivos electrónicos.
- **MIPI CSI-2** (Mobile Industry Processor Interface – Camera Serial Interface 2): Interfaz serie de alta velocidad para transmisión de datos de cámara.
- **MMCM** (Mixed-Mode Clock Manager): Bloque de gestión de reloj que genera frecuencias derivadas y controla la fase en dispositivos Xilinx.
- **PCB** (Printed Circuit Board): Placa de circuito impreso sobre la que se montan componentes electrónicos.
- **PCLK** (Pixel Clock): Señal de reloj que marca la cadencia de cada píxel transmitido por un sensor o procesador de vídeo.
- **PL** (Programmable Logic): Sección de lógica reconfigurable dentro de un dispositivo Zynq.
- **PS** (Processing System): Sección de procesador (ARM) y periféricos de un dispositivo Zynq.
- **RAW**: Formato de datos de imagen sin procesar proveniente del sensor.
- **RGB** (Red Green Blue): Espacio de color aditivo básico en vídeo e imagen.
- **SCCB** (Serial Camera Control Bus): Bus serie propietario de OmniVision compatible con I²C para configurar cámaras.

- **SLBM** (Stereo Local Block Matching): Algoritmo de correspondencia estéreo basado en búsqueda local por bloques.
- **SoPC** (System-on-Programmable Chip): Dispositivo que integra procesador y lógica programable en un único chip.
- **TPG** (Test Pattern Generator): Núcleo IP que genera patrones de vídeo sintéticos para prueba.
- **TMDS** (Transition-Minimized Differential Signaling): Codificación diferencial utilizada en HDMI y DVI.
- **TTL** (Transistor–Transistor Logic): Familia lógica y nivel de señal digital de 0-5 V.
- **UART** (Universal Asynchronous Receiver-Transmitter): Módulo serie asíncrono para comunicaciones.
- **USB** (Universal Serial Bus): Estándar de bus serie para conexión de periféricos.
- **VDMA** (Video Direct Memory Access): Núcleo AXI que mueve flujos de vídeo entre la PL y la memoria DDR.
- **VGA** (Video Graphics Array): Formato de vídeo analógico de 640×480 píxeles y, por extensión, conector VGA.
- **VTC** (Video Timing Controller): Núcleo IP que genera o detecta la temporización de vídeo.
- **VSYNC** (Vertical SYNC): Señal que indica el inicio de cada cuadro de vídeo.
- **YCbCr**: Formato digital de luminancia y crominancia derivado de YUV.
- **YUV**: Espacio de color que separa luminancia (Y) y crominancia (U,V).

CAPÍTULO 1. INTRODUCCIÓN

En este capítulo se expone la motivación del Trabajo Fin de Grado y el marco conceptual que lo sostiene. Además, se presentan los objetivos y la estructura del documento correspondiente a la memoria.

1.1. INTRODUCCIÓN

La necesidad de extraer información tridimensional del entorno se ha ido incrementando exponencialmente en las últimas décadas, especialmente en campos como la robótica, la inspección industrial, la realidad aumentada o la conducción autónoma. Este escenario ha dado lugar a aplicaciones cada vez más sofisticadas.

La estereopsis o visión estereoscópica [1] se presenta como una solución efectiva para estimar la profundidad de la escena a partir de dos o más vistas de la misma. Se basa en el cálculo de la disparidad entre las imágenes capturadas por dos cámaras desde puntos ligeramente diferentes, como se representa de forma esquemática en la Figura 1. Con estas imágenes convenientemente procesadas, es posible generar mapas de profundidad.

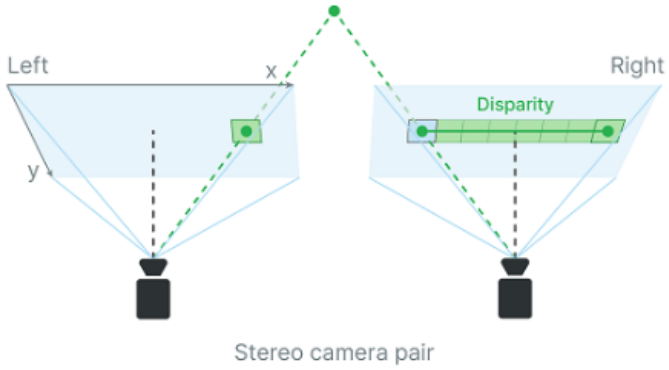

Figura 1. Profundidad desde la disparidad de las cámaras [2]

Por otro lado, el proceso de calibración de cámaras es el paso previo imprescindible para cualquier sistema estereoscópico, pues garantiza que las imágenes obtenidas estén alineadas y

libres de distorsiones. La rectificación geométrica [5] permite que, para un mismo punto en la escena, las coordenadas verticales de dicho punto en ambas imágenes coincidan, facilitando así el emparejamiento de píxeles entre cámaras (correspondencia estereoscópica). Esta tarea de calibración y rectificación puede realizarse *offline* con librerías consolidadas como OpenCV [6], o herramientas como Matlab, que implementan métodos basados en modelos geométricos.

De forma tradicional, la generación de mapas de disparidad puede abordarse con técnicas de coincidencia locales, como *block matching*, o con métodos más complejos (por ejemplo, *Graph Cuts*). La técnica de *block matching* [7] es muy popular en contextos de hardware programable por su relativa sencillez y capacidad de paralelización, algo especialmente valioso cuando se dispone de una FPGA para acelerar el procesamiento. La disponibilidad de metodologías basadas en síntesis de alto nivel (HLS), junto con bibliotecas específicas, como Vitis Vision [8], pueden simplificar la tarea de convertir algoritmos escritos en C/C++ (apoyados en librerías como OpenCV) en núcleos hardware específicos [9], lo que reduce el tiempo de desarrollo y facilita la optimización al ejecutarse directamente sobre la lógica programable.

En este Trabajo Fin de Grado (TFG) se propone el desarrollo de una plataforma *hardware/software* de estereopsis basada en SoPC [10] que hará uso de una placa de prototipado PYNQ-Z2 [11], que integra un APSoC (*All Programmable System-on-Chip*) Zynq-7000 de AMD/Xilinx además de dos cámaras OV5640 integradas en un array [12], que se conectarán mediante el protocolo SCCB (*Serial Camera Control Bus*) [13].

En la Figura 3 se representa el diagrama de bloques del funcionamiento del sistema que se propone desarrollar en el presente TFG:

- Calibración y rectificación de las cámaras de forma previa y *offline*.
- Captura y alineación de las imágenes en tiempo real sobre el dispositivo Zynq-7000.
- Aplicación del algoritmo de *stereo matching* (basado en *block matching*) para el cálculo del mapa de disparidad.
- Generación del mapa de profundidad (*depth map*) de la información relacionada con la distancia de las superficies de los objetos de la escena.

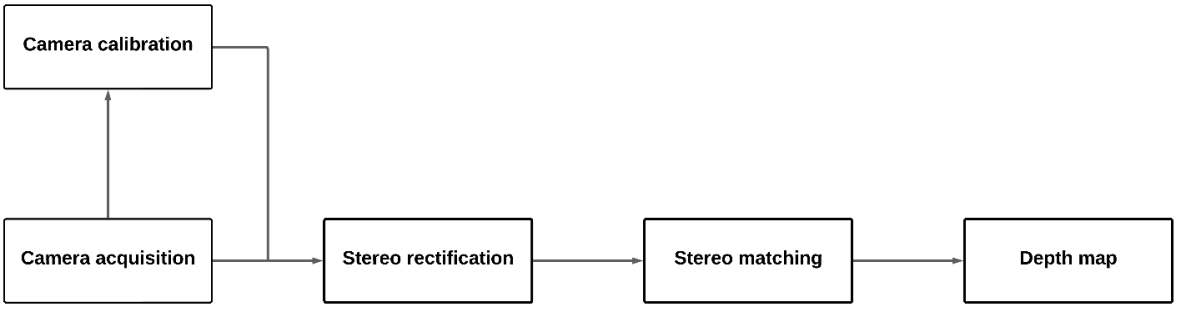

Figura 2. Diagrama de funcionamiento de la plataforma a desarrollar

La implementación de la arquitectura hardware se llevará a cabo en el entorno Vivado de AMD/Xilinx, para la descripción y conexión de los distintos IP. Por su parte, los procesos de calibración y rectificación se realizarán de manera *offline*, integrando los ajustes obtenidos en el proceso de *stereo matching*. La rutina de generación del bloque hardware asociado a la implementación del algoritmo SLBM (*Stereo Local Block Matching*), se desarrollará mediante Vitis HLS usando la biblioteca Vitis Vision de AMD, con la ayuda de Open CV como herramienta de verificación funcional para la correcta implementación del bloque IP.

En la actualidad, el continuo crecimiento de plataformas de hardware reconfigurable, como los denominados *System on Programmable Chip* (SoPC), ha facilitado la implementación de algoritmos de visión artificial de una forma eficiente y flexible. En particular, la familia Zynq-7000 de Xilinx [3], integra en un único encapsulado un procesador ARM y una lógica programable basada en FPGA (*Field Programmable Gate Array*), posibilitando la coexistencia de aceleradores hardware junto con el control y la ejecución de software de alto nivel. Este modelo *hardware/software* hace posible la implementación de sistemas con mayores prestaciones y menor consumo en tareas de procesamiento intensivo de datos, como es el caso del cálculo de la disparidad entre imágenes, en la implementación de sistemas de *stereo matching* [4].

1.2. OBJETIVOS

El objetivo de este TFG es el desarrollo de una plataforma de visión estereoscópica basada en SoPC, con la finalidad de generar información tridimensional de la escena en tiempo real. Este objetivo general se puede desglosar en los siguientes objetivos específicos:

O1: Desarrollar una IP para la integración del array de dos cámaras OV5640.

O2: Desarrollar una plataforma que permita la visualización de las imágenes capturadas a través de un monitor externo con conexión HDMI.

O3: Ejecutar la calibración intrínseca y extrínseca, junto con la rectificación geométrica del par de cámaras en Matlab. Exportar los parámetros resultantes y, a partir de ellos, diseñar un núcleo IP acelerado mediante Vitis HLS que implemente el algoritmo *Stereo Local Block Matching* (SLBM) utilizando la biblioteca Vitis Vision.

O4: Integrar y parametrizar el IP SLBM junto con el resto de núcleos de la plataforma, garantizando la comunicación entre módulos e interfaces del sistema.

O5: Generar el mapa de profundidad final y evaluar su rendimiento en tiempo real, analizando la precisión y la eficiencia del sistema resultante.

1.3. PETICIONARIO

El peticionario de este Trabajo Fin de Grado es la Escuela de Ingeniería de Telecomunicación y Electrónica (EITE) de la Universidad de Las Palmas de Gran Canaria (ULPGC), en calidad de institución pública que solicita la realización de dicho trabajo con el fin de superar los requisitos impuestos en la asignatura Trabajo Fin de Grado en el plan de estudios de la titulación Grado en Ingeniería en Tecnologías de la Telecomunicación (GITT).

1.4. ESTRUCTURA DE LA MEMORIA

La memoria se ha estructurado en siete capítulos que corresponden a la misma secuencia lógica que ha seguido el desarrollo del TFG:

Capítulo 1. Presenta el contexto y la motivación del trabajo, introduce los conceptos de visión estereoscópica y las tecnologías utilizadas, además de establecer los objetivos generales y específicos del TFG.

Capítulo 2. Describe la arquitectura *hardware* del APSoC Zynq-7000, y explica las interfaces disponibles en la placa de prototipado PYNQ-Z2 que se han utilizado para el registro y el procesado de las secuencias de vídeo capturadas por las cámaras.

Capítulo 3. Presenta del sensor OV5640, explicando su arquitectura interna, la secuencia de inicialización y la interfaz basada en el bus de control SCCB.

Capítulo 4. Describe el flujo de desarrollo Vivado / Vitis detallando el formato de vídeo utilizado (YUV 4:2:2) y la cadena completa de IPs (captura, *buffers*, *mixer*, HDMI) que permite validar en tiempo real las secuencias de vídeo capturadas por el array de cámaras sobre un monitor externo. Adicionalmente, se diseña la PCB adaptadora que facilita la integración del array de cámaras con la placa de prototipado PYNQ-Z2.

Capítulo 5. Profundiza en el algoritmo *Stereo Local Block Matching*, su implementación en lenguaje C/C++ mediante Vitis HLS, la creación del núcleo IP SLBM y su verificación funcional. Se explican tanto el acelerador y *testbench* como las métricas de rendimiento obtenidas.

Capítulo 6. Explica tanto el proceso del almacenamiento de las capturas de secuencias en una tarjeta SD, como la calibración y rectificación de las cámaras. También se justifica la integración del IP SLBM dentro de la cadena de captura de las secuencias de vídeo la creación del mapa de disparidad y profundidad, junto con las pruebas de precisión.

Capítulo 7. Resume los objetivos alcanzados, valora las limitaciones del sistema actual, e indica líneas de mejora y trabajos futuros.

CAPÍTULO 2. PLACA PYNQ-Z2 ARQUITECTURA E INTERFACES

En este capítulo, se describe la arquitectura APSoC Zynq-7000 y las interfaces clave disponibles en la placa de prototipado PYNQ-Z2 que harán posible la captura de las imágenes en el array de cámaras.

2.1. INTRODUCCIÓN A LA PLACA DE PROTOTIPADO PYNQ-Z2

Como se expuso en el capítulo anterior, la plataforma *hardware/software* desarrollada en el presente TFG se basa en la placa de prototipado PYNQ-Z2 de la empresa TUL [16] que integra un APSoC (*All Programmable System on Chip*) de la familia Zynq-7000 de AMD/Xilinx, que combina el procesador Dual Core Cortex-A9 de ARM con la lógica programable, basada en la arquitectura FPGA de la serie 7 (Artix™ 7 y Kintex™ 7).

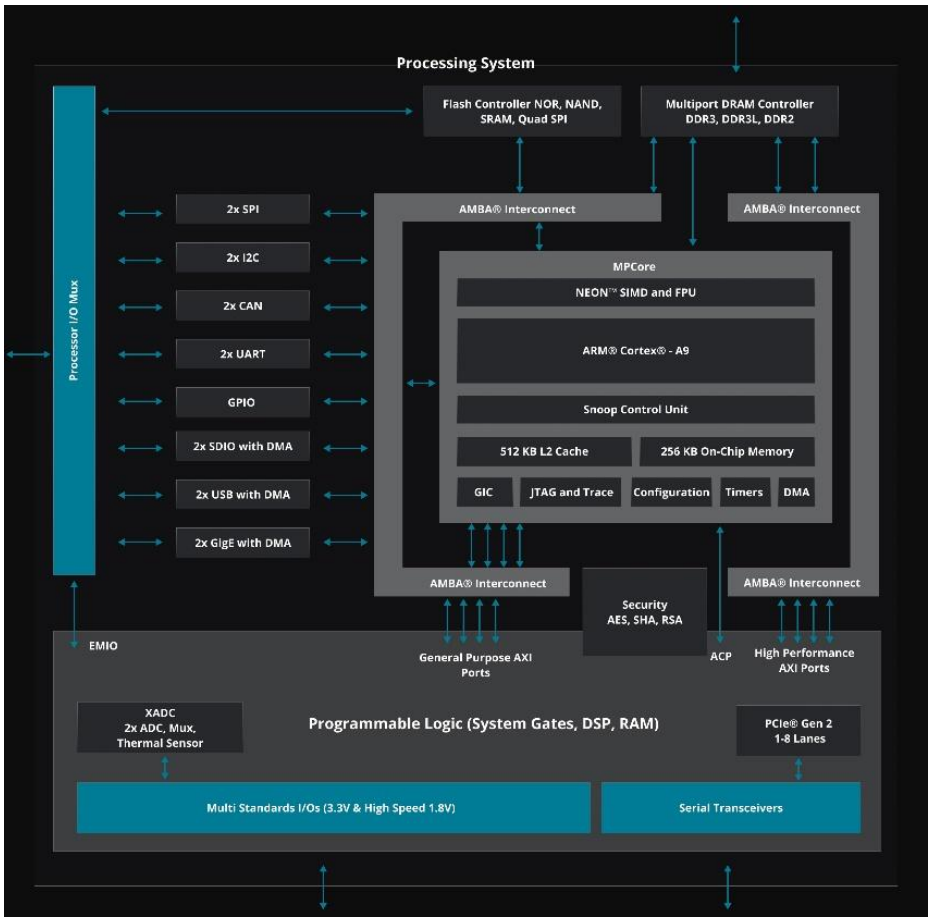

Figura 3. Arquitectura del AP SoC Zynq-7000 [14]

La Figura 3 sintetiza la arquitectura del APSoC Zynq de la serie 7 de AMD/Xilix. Muestra cómo el subsistema de procesamiento correspondiente a la sección PS que contiene dos núcleos ARM Cortex-A9, la memoria caché L2, los controladores de memoria DDR y el conjunto de periféricos estándar, se interconecta mediante buses AMBA con la lógica programable basada en FPGA. Dicha lógica FPGA, correspondiente a la sección PL, dispone de puertos AXI de propósito general y de alto rendimiento [15], así como de un puerto de coherencia, *Accelerator Coherency Port* (ACP), que hace posible un flujo de datos de baja latencia entre software y aceleradores hardware. Esta lógica contiene además un módulo denominado *Extended Multiplexed I/O* (EMIO), que facilitan llevar señalización adicional hacia la FPGA.

En concreto, la placa de desarrollo PYNQ-Z2 incorpora el dispositivo XC7Z020-1CLG400C de la familia Zynq-7000. Como se observa en la Figura 4, este dispositivo pertenece la gama *cost-optimized* y representa el escalón más alto del subgrupo inferior a la gama *mid-range*.

Zynq®-7000 SoC Family

		Cost-Optimized Devices						Mid-Range Devices			
	Device Name	Z-7007S	Z-7012S	Z-7014S	Z-7010	Z-7015	Z-7020	Z-7030	Z-7035	Z-7045	Z-7100
	Part Number	XC7Z007S	XC7Z012S	XC7Z014S	XC7Z010	XC7Z015	XC7Z020	XC7Z030	XC7Z035	XC7Z045	XC7Z100
Processing System (PS)	Processor Core	Single-Core ARM® Cortex™-A9 MPCore™ Up to 766MHz			Dual-Core ARM Cortex-A9 MPCore Up to 866MHz			Dual-Core ARM Cortex-A9 MPCore Up to 1GHz[1]			
	Processor Extensions	NEON™ SIMD Engine and Single/Double Precision Floating Point Unit per processor									
	L1 Cache	32KB Instruction, 32KB Data per processor									
	L2 Cache	512KB									
	On-Chip Memory	256KB									
	External Memory Support[2]	DDR3, DDR3L, DDR2, LPDDR2									
	External Static Memory Support[2]	2x Quad-SPI, NAND, NOR									
	DMA Channels	8 (4 dedicated to PL)									
	Peripherals	2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO									
	Peripherals w/ built-in DMA[2]	2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO									
	Security[3]	RSA Authentication of First Stage Boot Loader, AES and SHA 256b Decryption and Authentication for Secure Boot									
	Processing System to Programmable Logic Interface Ports (Primary Interfaces & Interrupts Only)	2x AXI 32b Master, 2x AXI 32b Slave 4x AXI 64b/32b Memory AXI 64b ACP 16 Interrupts									
Programmable Logic (PL)	7 Series PL Equivalent	Artix®-7	Artix-7	Artix-7	Artix-7	Artix-7	Artix-7	Kintex®-7	Kintex-7	Kintex-7	Kintex-7
	Logic Cells	23K	55K	65K	28K	74K	85K	125K	275K	350K	444K
	Look-Up Tables (LUTs)	14,400	34,400	40,600	17,600	46,200	53,200	78,600	171,900	218,600	277,400
	Flip-Flops	28,800	68,800	81,200	35,200	92,400	106,400	157,200	343,800	437,200	554,800
	Total Block RAM (# 36Kb Blocks)	1.8Mb (50)	2.5Mb (72)	3.8Mb (107)	2.1Mb (60)	3.3Mb (95)	4.9Mb (140)	9.3Mb (265)	17.6Mb (500)	19.2Mb (545)	26.5Mb (755)
	DSP Slices	66	120	170	80	160	220	400	900	900	2,020
	PCI Express®	—	Gen2 x4	—	—	Gen2 x4	—	Gen2 x4	Gen2 x8	Gen2 x8	Gen2 x8
	Analog Mixed Signal (AMS) / XADC[2]	2x 12 bit, MSPS ADCs with up to 17 Differential Inputs									
	Security[3]	AES & SHA 256b Decryption & Authentication for Secure Programmable Logic Config									
	Speed Grades: Commercial	-1			-1			-1			-1
	Speed Grades: Extended	-2			-2,-3			-2,-3			-2
	Speed Grades: Industrial	-1, -2			-1, -2, -1L			-1, -2, -2L			-1, -2, -2L

Figura 4. Familia de dispositivos Zynq®-7000 SoC

La diferencia esencial con respecto a los demás modelos se encuentra en la sección de la lógica programable (PL). El dispositivo Z-7020 ofrece entre otros recursos 85k celdas lógicas, 52k *Look-Up Tables* (LUTs), y 106k Flip-Flops, cruciales para la realización de este proyecto. También incluye 4.9 Mb de BRAM (*Block RAM*) distribuidos en 140 bloques de 36kb y 220 slices DSP (*Digital Signal Processor*), suficientes para la implementación de aceleradores con cierto paralelismo [17].

2.2. INTERFACES DE LA PLACA DE DESARROLLO PYNQ-Z2

La placa de desarrollo TUL PYNQ-Z2 se puede alimentar a través del puerto Micro-USB que incorpora, ajustando el jumper J9 en la posición USB. También puede alimentarse a través de una fuente externa de 12V mediante el conector DC1, o una batería cuyo terminal positivo se conectaría al pin VIN del conector Arduino J7 y el negativo al pin GND en el caso de requerir más energía. Adicionalmente proporciona un oscilador que genera una señal de 50MHz en el pin PS_CLK, usado para generar las señales de reloj del subsistema PS que puede operar hasta un máximo de 650MHz.

Esta placa de prototipado proporciona además un conjunto completo de interfaces que integran múltiples periféricos externos. En este TFG se utilizarán los siguientes:

- Micro-USB: Dispone de USB 2.0 para alimentar la placa y mediante un puente FT2232HQ, se convierte USB a UART TTL para la comunicación.
- Conectores Raspberry Pi: Contiene 40 pines mediante los que se conectará el array de cámaras OV5640 con la placa de prototipado PYNQ-Z2.
- Ranura MicroSD: Tiene una interfaz SD 3.0 FAT32 que se utilizará para almacenar los *videobuffers* crudos en formato BIN de las secuencias de vídeo capturadas por las cámaras.
- Puerto HDMI: Con salida TMDS, se trasmite vídeo hacia una pantalla externa para monitorizar en tiempo real el procesamiento del sistema.
- Led0: Se utiliza como una señal simple de control para verificar que los buffers asociados a cada cámara entren correctamente en el bucle de la aplicación *software* de la plataforma.

– DDR: Se emplea como memoria principal del sistema, alojando tanto las aplicaciones en ejecución de la sección PS, como los *videobuffers* crudos transferidos desde la sección PL.

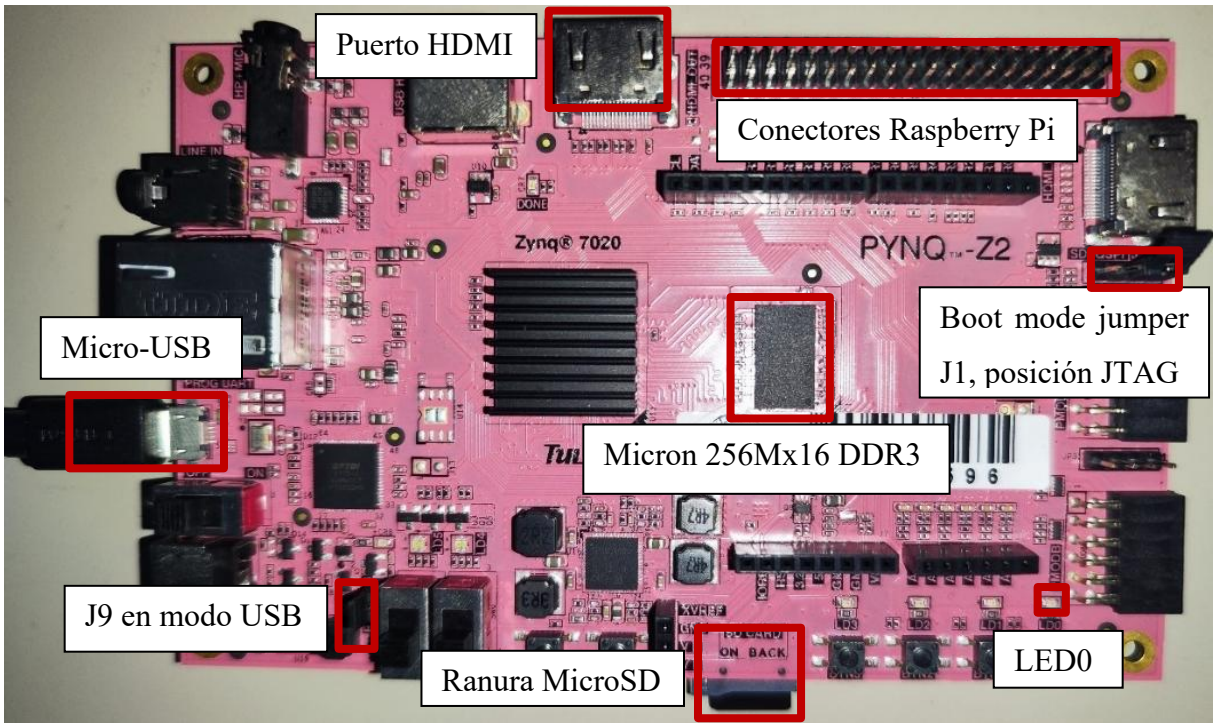

Figura 5. Periféricos externos de la placa de prototipado PYNQ-Z2

La Figura 5 muestra la disposición de los periféricos externos de la placa de prototipado PYNQ-Z2. En ella se pueden ver los pines del conector Raspberry Pi donde se conectará el array de cámaras OV5640 como un periférico estándar.

CAPÍTULO 3. ARRAY DE CÁMARAS OV5640

En este capítulo se justifica la elección del sensor y se describe el proceso de inicialización y de control del protocolo SCCB, así como el diseño de la PCB que facilite la integración de los arrays de cámaras considerados en el presente TFG con la placa de prototipado PYNQ-Z2.

3.1. INTRODUCCIÓN AL SENSOR OV5640

OmniVision OV5640 es un sensor de imágenes CMOS de 5 Megapíxeles que dispone de una matriz de imágenes de 2624 columnas por 1964 filas (5.153.536 píxeles). La Figura 6 muestra una sección transversal de la matriz del sensor de imagen.

Los filtros de color están dispuestos en un patrón Bayer. La matriz de colores primarios BG/GR está organizada de forma alternada por líneas. De los 5,153,536 píxeles, 5,038,848 (2592x1944) son píxeles activos y pueden ser utilizados como salida. Los demás píxeles se usan para la calibración del nivel de negro y la interpolación.

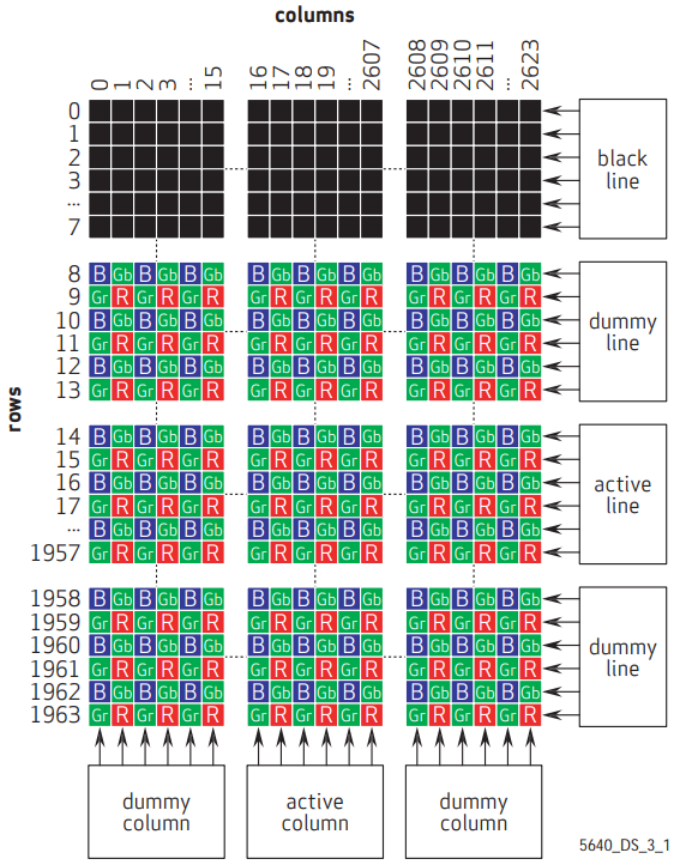

Figura 6. Filtros de color de la región de la matriz de sensores [18]

3.2. ARQUITECTURA DEL SENSOR OV5640

El núcleo del sensor dispone de una matriz de píxeles óptico con un formato de 2.624 × 1.964 píxeles y un tamaño de píxel 1.4 µm, así como circuitería analógica para el muestreo en columnas, amplificación programable y un ADC de 10 bits por píxel. Este núcleo proporciona los datos de imagen en flujo continuo a una tasa de fotograma constante, sincronizado a través de señales HREF (referencia de línea horizontal) y VSYNC (sincronismo vertical).

El sensor emplea *rolling shutter*, un método de captura en el que el obturador expone la escena mediante un barrido progresivo. De este modo, la fotografía o el vídeo resultante no reflejan toda la escena en un único instante, sino que se escanean línea a línea, de forma vertical u horizontal, para componer la imagen [19]. También controla el tiempo de exposición mediante compensación de ajuste del intervalo de muestreo para cada fila.

El sensor OV5640 cuenta con dos interfaces de salida de video: una interfaz paralela DVP (*Digital Video Port*) de 8/10 bits, y una interfaz serie MIPI CSI-2 doble canal (*dual-lane*) para alta tasa de bits. A través de estas interfaces, el sensor puede suministrar flujo de vídeo en varios formatos, entre los que se encuentran RAW10/8 (Bayer), YUV/ YCbCr4:2:2, RGB565/555/444, o el estándar CCIR656, así como imágenes JPEG comprimidas en hardware. Es capaz de generar hasta 60 fotogramas para imágenes 720p, y 90 fotogramas para resolución VGA como se recoge en la Tabla 1.

Tabla 1. Formato y tasa de fotogramas.

format	resolution	frame rate	scaling method	pixel clock
5 Mpixel	2592x1944	15 fps	full resolution (dummy 16 pixel horizontal, 8 lines) 2608x1952 with dummy	96/192 MHz
1280x960	1280x960	45 fps	subsampling in vertical and horizontal 1296x968 supports 2x2 binning	96/192 MHz
1080p	1920x1080	30 fps	cropping from full resolution 1936x1088 with dummy pixels	96/192 MHz
720p	1280x720	60 fps	cropping 2592x1944 to 2560x1440 subsampling in vertical and horizontal 1296x728 with dummy supports 2x2 binning	96/192 MHz
VGA	640x480	90 fps	subsampling from 1280x960 648x484 with dummy supports 2x2 binning	48/96 MHz
QVGA	320x240	120 fps	subsampling from 1280x960 324x242 with dummy supports 2x2 binning	24/48 MHz

La configuración completa y el control del sensor (por ejemplo, elección del formato, resolución, parámetros ISP), se realizan mediante un bus de control serie basado en el protocolo SCCB (con compatibilidad con I²C) para acceder a un banco interno de registros. En aplicaciones estereoscópicas es esencial configurar y sincronizar los dos sensores OV5640 de manera idéntica.

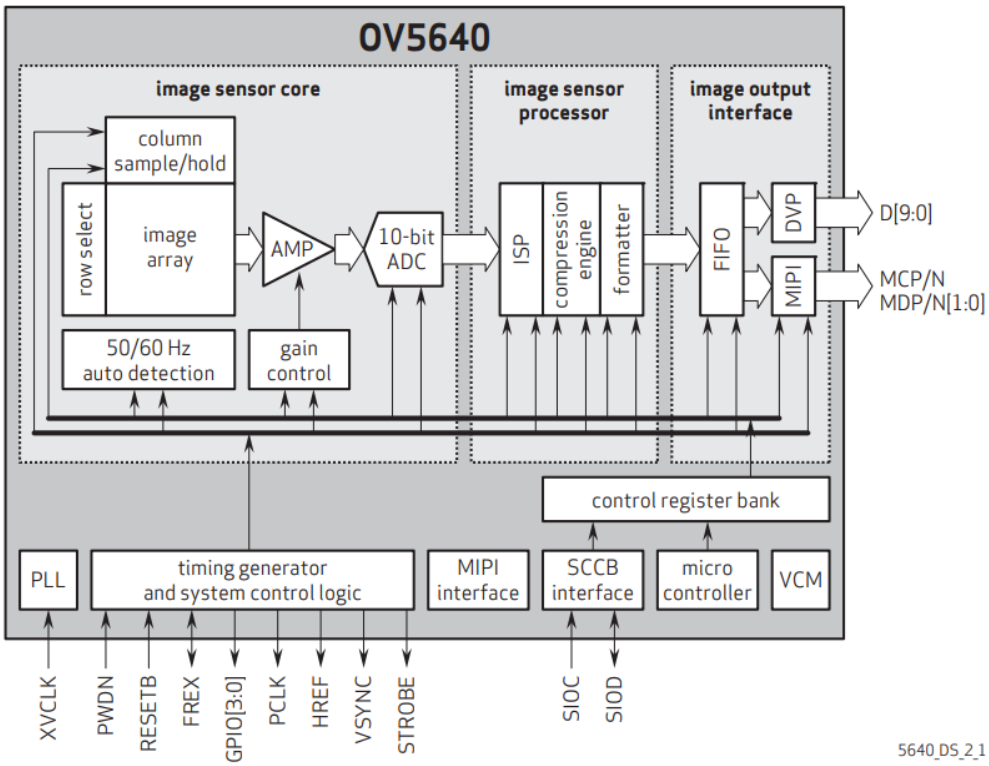

Figura 7. Diagrama de bloques del OV5640

Por otro lado, el sensor OV5640 genera una señal VSYNC para cada fotograma y HREF para cada línea válida, además de una señal PCLK (reloj de píxel) para muestrear los datos de imagen (Figura 7). Si se requiere el ajuste de temporización de fotogramas, como se establece en la Tabla 2, los registros 0x380C y 0x380D definen el número de píxeles por línea (HTS, determina en la frecuencia de la señal PCLK) y 0x380E y 0x380F definen el número de líneas por fotograma (VTS, determina la duración de los fotogramas). La sincronía en la tasa de fotogramas se asegura manteniendo ambos sensores con el mismo HTS/VTS.

Como se observa en la Figura 7, la interfaz de entrada/salida del sensor OV5640 engloba tanto la captura de imagen por el bus DVP paralelo o CSI-2 (MIPI), como la configuración y control del sensor:

- XCLK: reloj maestro que alimenta el sensor.
- PCLK (*Pixel Clock*): sincroniza la transferencia de cada dato de píxel.
- VSYNC: señal de sincronismo de fotograma; genera un flanco al inicio de cada *frame*.

- HREF: indica las líneas de datos válidas dentro de cada fotograma.
- D[9:0]: bus paralelo de datos de imagen (10 bits) en modo DVP.
- MIPI_D0_P/N, MIPI_D1_P/N: pares diferenciales para la salida CSI-2 de alto rendimiento.
- SIOC, SIOD: líneas de reloj y datos del bus SCCB para lectura/escritura de registros internos.
- RESET_B y PWDN: señales de reinicio asíncrono y modo de bajo consumo.
- FREX y STROBE: entrada de disparo externo y señal de control de *flash*.
- VCM: control de tensión para la bobina de enfoque automático de la lente.

Mediante esta combinación de señales de reloj, datos, sincronismo y configuración, el sensor OV5640 puede operar en modo vídeo o fotografía, empleando la interfaz DVP para implementaciones sencillas y la CSI-2 (MIPI) para anchos de banda y prestaciones superiores.

Tabla 2. Registros de delimitación de imagen

address	register name	default value	R/W	description
0x3809	TIMING DVPHO	0x20	RW	Bit[7:0]: DVP output horizontal width[7:0] low byte
0x380A	TIMING DVPVO	0x07	RW	Bit[2:0]: DVP output vertical height[10:8] high byte
0x380B	TIMING DVPVO	0x98	RW	Bit[7:0]: DVP output vertical height[7:0] low byte
0x380C	TIMING HTS	0x0B	RW	Bit[3:0]: Total horizontal size[11:8] high byte
0x380D	TIMING HTS	0x1C	RW	Bit[7:0]: Total horizontal size[7:0] low byte
0x380E	TIMING VTS	0x07	RW	Bit[7:0]: Total vertical size[15:8] high byte
0x380F	TIMING VTS	0xB0	RW	Bit[7:0]: Total vertical size[7:0] low byte
0x3810	TIMING HOFFSET	0x00	RW	Bit[3:0]: ISP horizontal offset[11:8] high byte
0x3811	TIMING_HOFFSET	0x10	RW	Bit[7:0]: ISP horizontal offset[7:0] low byte
0x3812	TIMING VOFFSET	0x00	RW	Bit[2:0]: ISP vertical offset[10:8] high byte
0x3813	TIMING VOFFSET	0x04	RW	Bit[7:0]: ISP vertical offset[7:0] low byte

3.2.1. Secuencia de inicialización del sensor

Para la correcta inicialización de la cámara OV5640 se ha de seguir un conjunto de secuencias de encendido (*power-up*), que garantiza la estabilidad del sensor y protección de los circuitos internos (Figura 8).

1. Activar DOVDD y AVDD: Inicialmente, se aplican los voltajes para las entradas/salidas (DOVDD) y para la parte analógica (AVDD). Es posible aplicarlos al mismo tiempo, aunque, si no, DOVDD debe preceder a AVDD.
2. Desactivar PWDN: Tras esperar al menos 5 ms desde que AVDD se estabiliza (t_2), PWDN se establece a un estado bajo para desactivar el modo de apagado.
3. Desactivar Reset: Tras un tiempo de espera de un milisegundo (t_3), se desactiva el Reset llevándolo a un estado alto, lo que permite al sensor iniciar su proceso de arranque interno.
4. Inicializar mediante SCCB: Transcurridos 20 ms (t_4), el bus SCCB (equivalente al bus I^2C) ya está disponible, y el sistema principal puede comenzar a escribir los registros para el ajuste de los parámetros de la cámara OV5640.

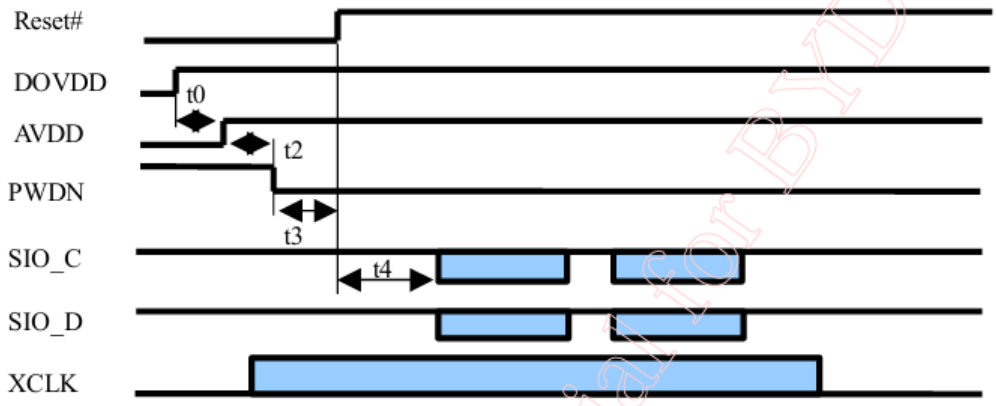

Figura 8. Secuencia de power-up

En la Figura 9 se ilustra cómo se conecta el sensor OV5640 (U1) en un módulo de cámara, incluyendo sus fuentes de energía, líneas para datos y control, además de filtros y reguladores.

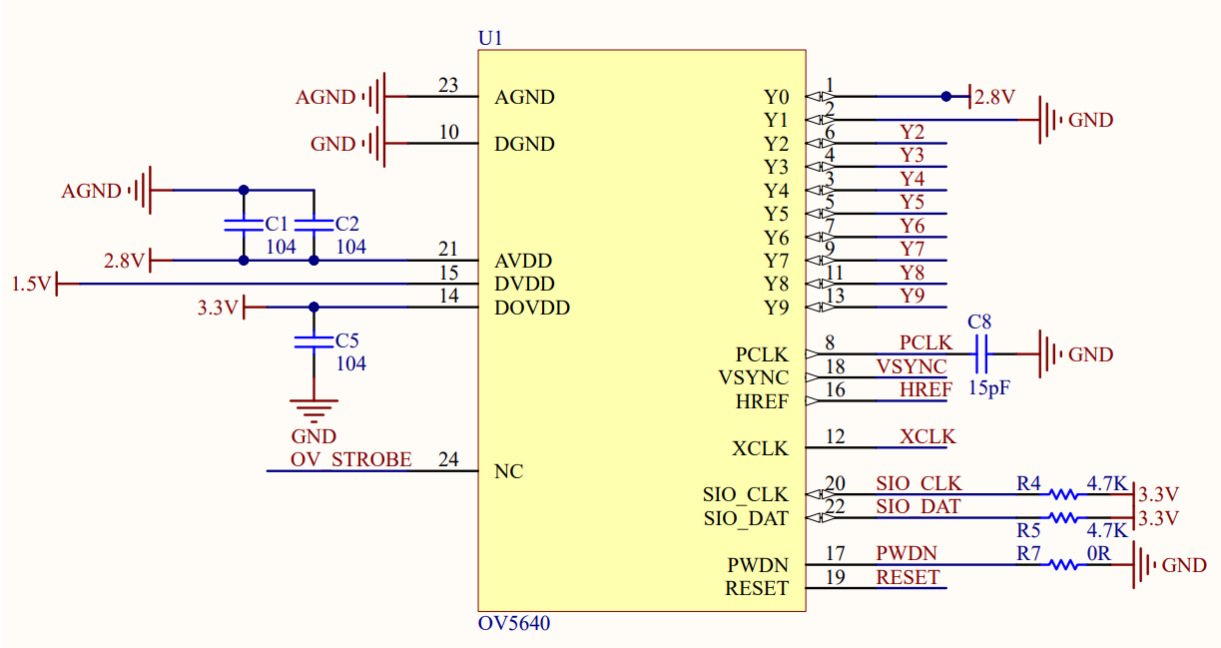

Figura 9. Esquema eléctrico de la interfaz y alimentación del sensor CMOS OV5640 [20]

Los buses SIO_CLK y SIO_DAT deben tener resistencias de pull-up externas; el valor típico de estas resistencias es de 4.7K-5.1K. La señal de RESET está activa a nivel bajo y cuenta con una resistencia de pull-up interna. PWDN está activa a nivel alto y tiene una resistencia de pull-down interna. Tanto la señal RESET como PWDN deben ser controladas externamente para una secuencia de encendido adecuada.

3.2. INTERFAZ DE CONTROL SCCB

La interfaz SCCB (*Serial Camera Control Bus*) es un bus serie de dos hilos, SIOC (reloj) y SIOD (datos), utilizado por OmniVision para acceder a los registros internos del sensor OV5640. Aunque guarda similitudes con el protocolo I²C (mismos ciclos de inicio, parada y ACK), la interfaz SCCB define cada fase de transferencia como un byte: primero la dirección de registro (8 bits) y luego el dato (8 bits) [21]. Mediante esta secuencia de *start*, dirección, dato y *stop*, el controlador puede configurar parámetros como la exposición, ganancia, formato de imagen, entre otras funciones del sensor.

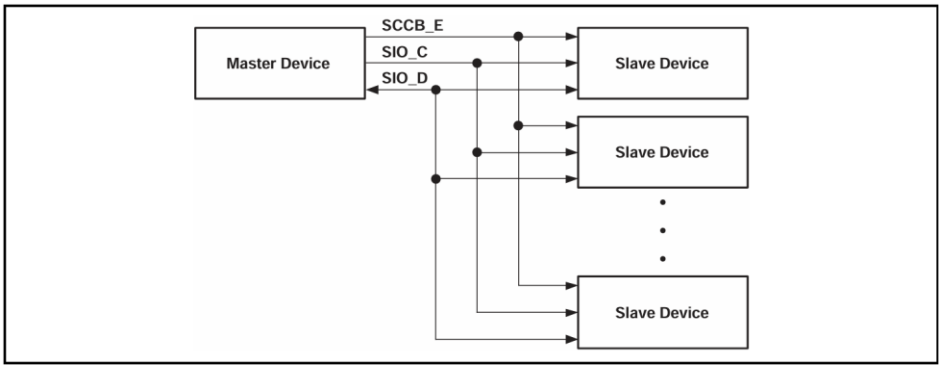

Figura 10. Diagrama de bloques funcional SCCB

Los sensores de OmniVision solo funcionarán como dispositivos esclavos y la interfaz *backend* complementaria debe actuar como maestro. La implementación de 3 hilos permite que un dispositivo SCCB maestro se conecte a múltiples dispositivos esclavos, como se muestra en la Figura 10.

Una fase contiene un total de 9 bits. Los 9 bits consisten en una transmisión secuencial de datos de 8 bits seguida por un noveno bit (ver .

Figura 11). El noveno bit es un bit de *Don't-Care* o un bit NA, dependiendo de si la transmisión de datos es de escritura o de lectura. El número máximo de fases que se pueden incluir en una transmisión es de tres, y en cada fase siempre transmite primero el bit más significativo (MSB).

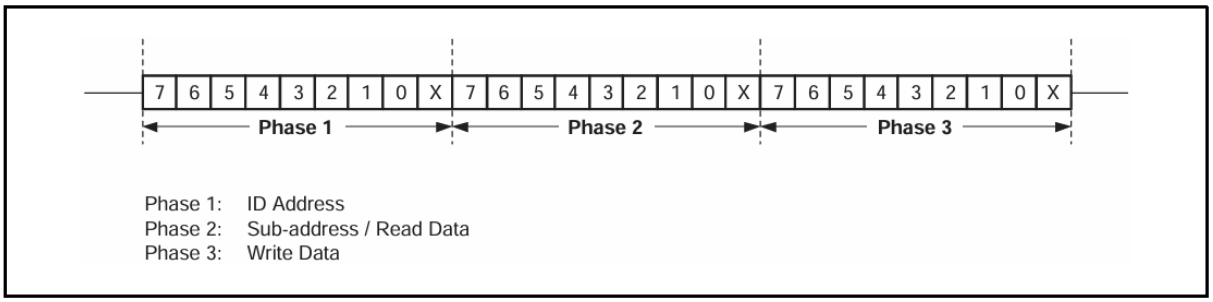

Figura 11. Fases de transmisión SCCB

El protocolo SCCB contiene solo 3 tipos de ciclos de transmisión:

1. Ciclo de escritura en 3 fases (*Device Addr* + *Register Addr* + *Data*)

Tras el START, el maestro envía la dirección del dispositivo con el bit R/W = 0 y espera el ACK; a continuación se transmite la dirección del registro interno seguida de otro ACK y, por último, el byte de datos con su ACK final antes del STOP (Figura 12). Este ciclo completo es el más frecuente, permitiendo programar cualquier registro arbitrario del sensor en una sola transacción.

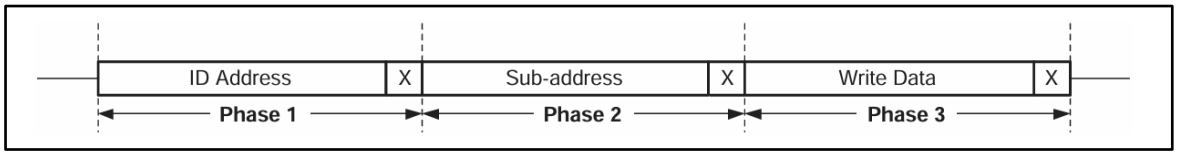

Figura 12. Ciclo de transmisión de escritura en 3 fases

2. Ciclo de escritura en 2 fases (Device Addr + Register Addr)

En este caso el maestro solo coloca la dirección de dispositivo (R/W = 0) y, tras el ACK, el registro destino; después emite STOP. Sirve para pre-cargar el puntero interno del sensor cuando, inmediatamente después, se va a lanzar una operación de lectura aleatoria. El maestro repetirá START y enviará ahora la dirección del mismo dispositivo con R/W = 1 para recibir el contenido del registro apuntado (Figura 13).

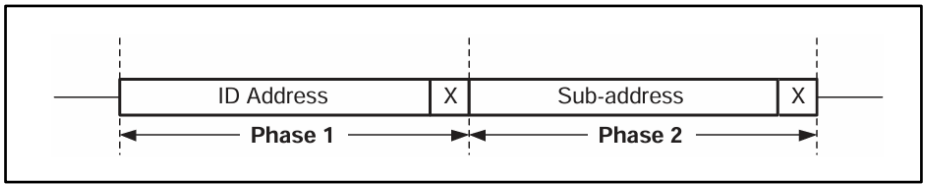

Figura 13. Ciclo de transmisión de escritura en 2 fases

3. Ciclo “rápido” en 2 fases (*Device Addr + Data*)

Cuando el puntero interno del sensor ya está correctamente posicionado, ya sea porque el sensor lo autoincrementa o porque se empleó previamente el ciclo anterior, basta con un START, la dirección de dispositivo (R/W = 0), ACK, el nuevo dato, ACK y STOP (Figura 14). De esta forma se pueden volcar secuencias largas de configuración minimizando el *overhead* del bus.

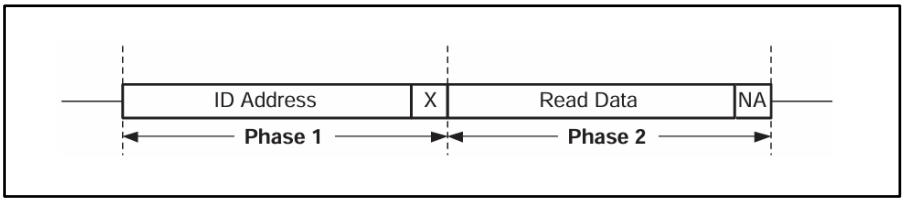

Figura 14. Ciclo de transmisión de lectura en 2 fases

3.3. ARRAY DE CÁMARAS OV5640

3.3.1. Disposición física del array

A lo largo del desarrollo de la plataforma implementada en el presente TFG, se ha verificado el correcto funcionamiento del sistema sobre dos *arrays* de cámaras de distintos fabricantes. A continuación, se describen los modelos de placas comerciales utilizados.

3.3.1.1. Modelo PZ5640-D

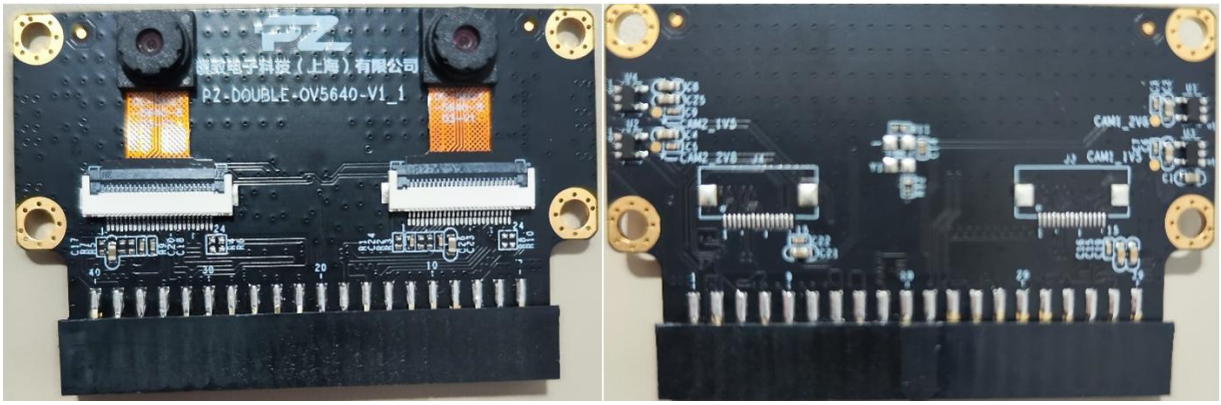

Figura 15. Módulo dual de cámaras OV5640 (placa PZ-DOUBLE-OV5640-V1.1)

Este modelo de referencia PZ5640-D (Figura 15) del fabricante Pú zhì, está compuesto por dos módulos de cámaras OV5640 con autoenfoque, montados simétricamente sobre una PCB de alimentación y señalización común.

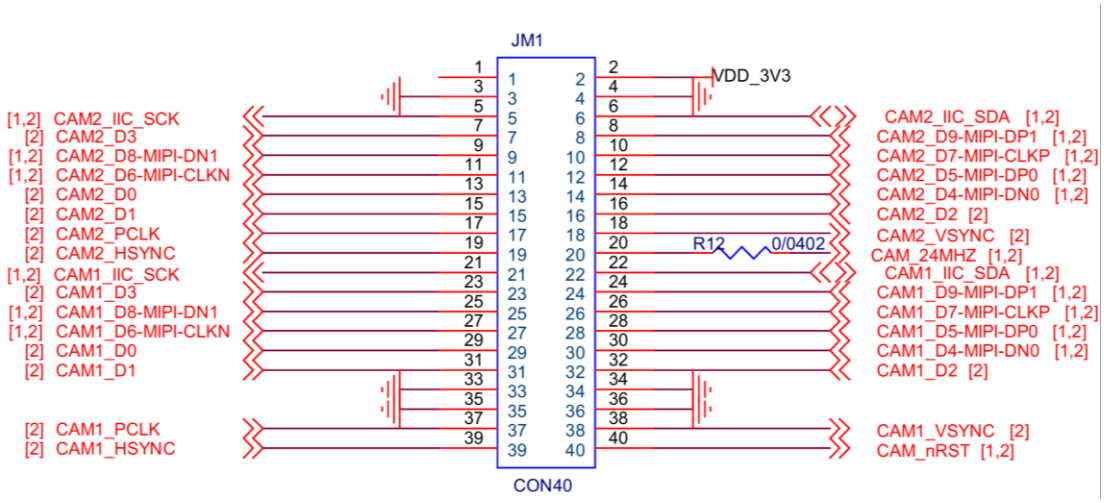

Figura 16. Esquemático del conector JM1

La conexión de este array con la placa de prototipado PYNQ-Z2 se realiza mediante un conector hembra JM1 de 40 pines, dispuesto en el lado inferior a los módulos de cámara (Figura 16). Este modelo no incluye un oscilador que permita disponer de la señal XCLK con una frecuencia de referencia de 24 MHz, por lo que debe suministrarse desde una fuente externa.

Cada sensor expone las 10 líneas de datos D9 a D0 pero, como en la placa de prototipado PYNQ-Z2 únicamente se aceptan 8 bits, se descartan los 2 bits más significativos del sensor para ajustar el ancho de bus físico sin repercusión en la validez de las imágenes capturadas.

3.3.1.1. Modelo ATK-OV5640-DUAL

El segundo dispositivo de referencia es el modelo ATK-MC5640D V1.91 (Figura 18), también diseñado para configurar un par estéreo con los mismos sensores OV5640, aunque en ese caso con enfoque manual.

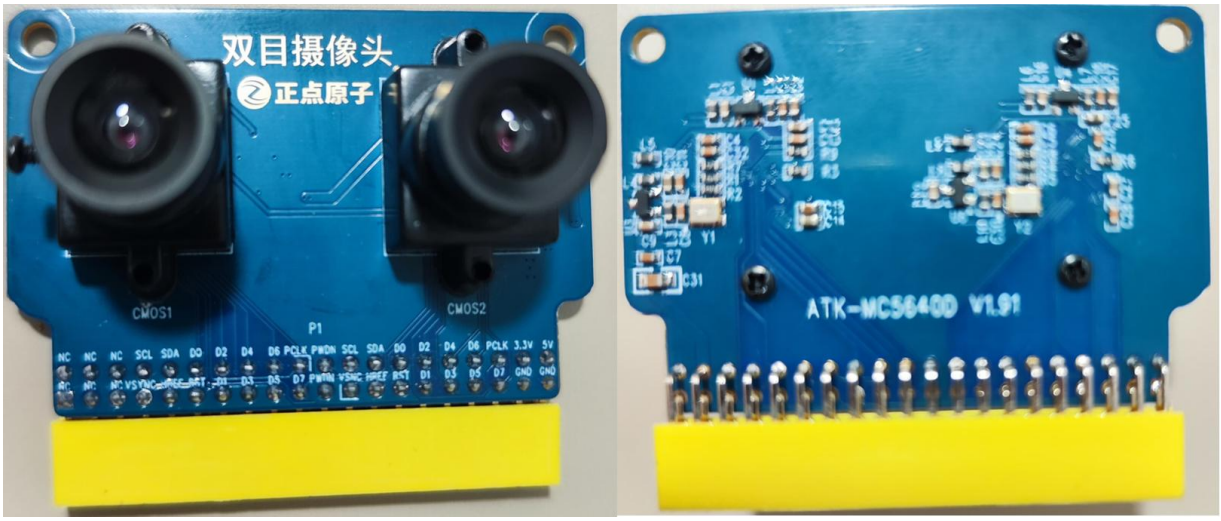

Figura 17. Módulo dual de cámaras OV5640 (placa ATK-MC5640D (V1.91))

En su topolgía de bus emplea un *header* macho de 2 × 20 pines con una numeración física similar al conector JM1 del modelo PZ5640-D. Sin embargo, la versión ATK-OV5640-DUAL integra de fábrica un oscilador de 24 MHz sobre la propia tarjeta.

Se respeta la misma distribución de masas, alimentación 3V3 y líneas de control RESET/PWDN (Figura 18), por lo que el firmware escrito para el modelo PZ5640-D será reutilizable sin cambios, excepto el registro que habilita la salida de reloj interno durante la secuencia de *power-up*.

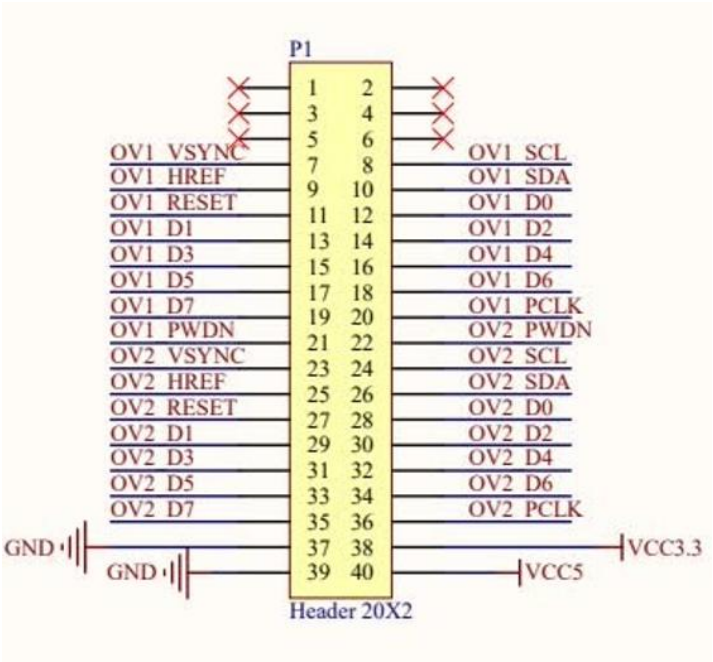

Figura 18. Esquemático del Header ATK-OV5640-DUA

CAPÍTULO 4. INTEGRACIÓN DE LA PLATAFORMA DE VISUALIZACIÓN PARA LAS CÁMARAS OV5640

En este capítulo se describe el flujo de desarrollo Vivado/Vitis empleado para la creación de la cadena captura-procesado-visualización de la secuencia de vídeo capturada por las cámaras integradas en el array. Adicionalmente, se justifica el formato de vídeo utilizado y se describen los núcleos IP más importantes del diseño de la arquitectura hardware de la plataforma. Se finaliza el capítulo con la validación funcional de las cámaras OV5640, mostrando las imágenes de ambas.

4.1. INTRODUCCIÓN AL FLUJO DE DISEÑO VIVADO 2023.1 / VITIS 2023.1

Para el desarrollo de la arquitectura *hardware* se usa el entorno de desarrollo para SoCs y FPGA proporcionado por AMD/Xilinx Vivado Design Suite en su versión 2023.1. Este entorno incluye herramientas de entrada de diseño, síntesis, *placement/routing* y verificación/simulación. Las funciones avanzadas de este entorno ayudan a reducir en el diseño HW los tiempos de compilación, así como agilizar las iteraciones de diseño y mejorar la precisión de la estimación de consumo de energía para los SoC adaptativos y FPGA de AMD/Xilinx [22].

Las herramientas de Vitis funcionan en conjunto con AMD/Xilinx Vivado Design Suite a fin de proporcionar un nivel más alto de abstracción del diseño. En el desarrollo del presente TFG, se han utilizado las siguientes herramientas del entorno de desarrollo SW Vitis:

- Vitis Embedded: para desarrollar el código de la aplicación C/C++ que se ejecutarán en el procesador ARM integrado.
- Vitis HLS: para convertir funciones C/C++ en bloques IP RTL optimizados para FPGA. Permite introducir directivas (pragmas) para explorar arquitecturas de rendimiento y generar código RTL que se integre con Vivado/Vitis.
- La biblioteca de código abierto Vitis Vision, que proporciona algoritmos de visión por computador en lenguaje C/C++ altamente optimizados para síntesis en FPGA,

facilitando la implementación eficiente de *kernels* de procesamiento de imágenes directamente en lógica programable.

4.1.1. Flujo de desarrollo bare-metal

Como se muestra en la Figura 19, el flujo de desarrollo bare-metal en los dispositivos Zynq-7000 de AMD/Xilinx, consta de tres etapas:

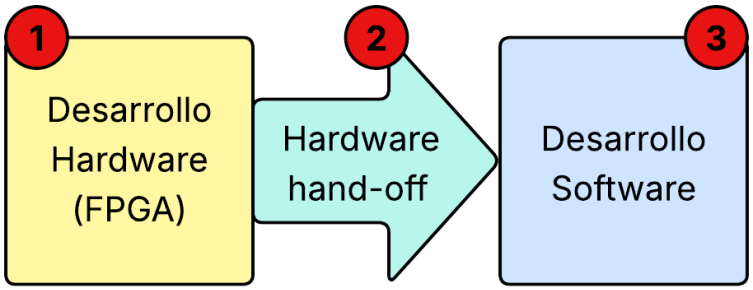

Figura 19. Etapas del flujo de desarrollo bare-metal

1. Desarrollo HW (*Hardware Development*): Se diseña la arquitectura *hardware* de la plataforma que se implementará en la sección de la lógica programable (PL) del dispositivo, por lo general a partir de núcleos IP. En esta etapa se sintetiza y se implementa el diseño de la arquitectura HW, generándose un fichero con el *bitstream* (.bit) que determinará la configuración de los recursos disponibles en PL.
2. Transferencia HW (*Hardware Hand-off*): Es esta etapa se genera un fichero que contiene la información asociada al diseño de la arquitectura HW de la plataforma, incluyendo el *bitstream* y los parámetros de inicialización de PS, así como información relativa a los periféricos de E/S y las direcciones de memorias asignadas.
3. Desarrollo SW (*Software Development*): Es esta etapa se importa el entorno de desarrollo SW Vitis, el fichero que contiene el diseño de la plataforma HW, generándose el BSP (*Board-Support Package*) a partir del cual se implementará la aplicación SW.

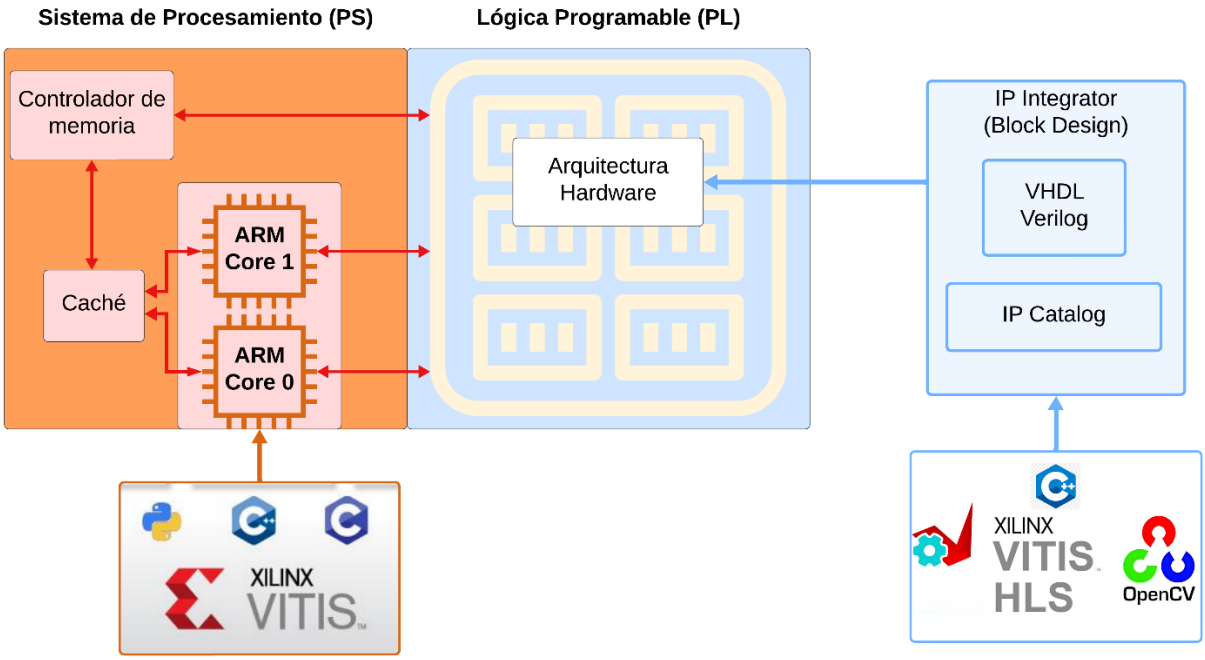

Figura 20. División hardware/software en un Zynq-7000

La Figura 20 presenta el esquema clásico de diseño para los APSoC Zynq-7000, donde se define con precisión qué sección del algoritmo se procesa en los núcleos ARM (Cortex-A9) y cuál se implementa en la lógica programable (PL).

A la izquierda, el entorno del procesador (PS), señalado en rojo, reúne la memoria caché, el controlador DDR y los dos ARM, que ejecutan software en lenguaje C/C++ compilado con Vitis Embedded. Las flechas anaranjadas indican los puertos AXI que enlazan PS con PL. Haciendo uso de estos, el software ajusta registros de control (*AXI-Lite*) y desplaza grandes volúmenes de datos vía DMA (AXI-HP/HPC), permitiendo que ambos entornos usen la memoria del sistema con latencias definidas.

En el lado derecho, la zona azul simboliza la lógica programable (PL) y, por tanto, todo el entorno de diseño que Vivado pone a disposición del diseñador/a. En esta sección se encuentra el motor de aceleración *hardware*, que se sintetiza a partir de un *kernel* C/C++ gracias a Vivado HLS. Tras su creación, este IP pasa a formar parte del IP Catalog y, ya dentro de Vivado IP Integrator, se enlaza con bloques propios y comunes (VDMA, *Video Mixers*, PHY HDMI, etc.)

y con la infraestructura de reloj, *resets*, restricciones y rutas AXI que también se especifican en el entorno Vivado. De esta forma, se completa la partición HW/SW.

El resultado es un *bitstream*, junto al BSP y la aplicación ejecutable, que cooperan por medio de interfaces AXI, todo ello administrado dentro del proceso completo de Vivado/Vitis para optimizar al máximo los recursos, la latencia y el ancho de banda en en el sistema integrado de visión propuesto.

4.1.2. Interfaz AXI en el flujo *bare-metal*

La familia AXI (*Advanced eXtensible Interface*) es la especificación de la interfaz AMBA de Arm® [23]. La tercera generación de este protocolo AXI3 es con la que se comunican las secciones *Processing System* (PS) y *Programmable Logic* (PL) en los APSoC Zynq-7000 mientras que, en esta última se utiliza por lo general AXI4. Entender los conceptos AXI resulta imprescindible para seguir la plataforma desarrollada en este TFG.

El bus AXI se define según un modelo de transacciones, el cual estructura la comunicación entre un dispositivo *Manager* (se suele denominar *Master*) y un dispositivo *Subordinate* (*Slave*). En conformidad con el modelo de transacciones, el bus AXI está dividido en cinco canales de forma lógica independiente, cada uno de los cuales tiene su propia lógica con su propio conjunto de señales [24].

- Canal de escritura - prefijo AW.
- Canal de escritura - prefijo W.
- Canal de escritura - prefijo B.
- Canal de lectura - prefijo AW.
- Canal de lectura - prefijo R.

Los canales de petición (AW y AR) llevan la información de control que especifica la transacción (dirección, longitud de ráfaga, atributos de caché...), mientras que los canales de datos (W y R) transportan la carga útil. Tras una transferencia de escritura, B devuelve al *Manager* un código de finalización, así como posibles errores que certifiquen el hecho de que la transferencia se haya finalizado. En términos funcionales:

- Para la transferencia de escritura, el dispositivo *Manager* transmite por el canal AW la cabecera de la transacción (dirección de destino, longitud de la ráfaga y resto de

parámetros de control); inmediatamente después, envía la carga útil por el canal W, palabra a palabra, hasta completar la ráfaga especificada. Posteriormente, el dispositivo *Subordinate* confirma la operación mediante una respuesta por B.

- En la transferencia de lectura, el dispositivo *Manager* emite la petición en AR y posteriormente el dispositivo *Subordinate* devuelve los datos solicitados en la línea de datos R.

4.1.2.2 AXI4-Full

AXI4-Full es la versión de AMBA diseñada para las transferencias de memoria de alto rendimiento; la documentación correspondiente la caracteriza como una *interface memory-mapped* que soporta tráfico a ráfagas. Este tipo de transacción se separa según los cinco canales lógicos independientes, donde la dirección, los datos y las respuestas pueden superponerse sin bloquearse a sí mismas. Dado que se codifica de 0 a 255, el protocolo puede gestionar ráfagas de 1 a 256 *beats*, y alcanzando anchos de banda a nivel de *gigabyte* en los SoC modernos [25], [26]. En terminología AMBA-AXI, un *beat* es la transferencia individual de una palabra de datos que tiene lugar dentro de una ráfaga (*burst*).

4.1.2.2 AXI4-Lite

AXI4-Lite constituye un subconjunto simplificado de *AXI4-Full*: todas sus transacciones son de un solo *beat*, el tamaño de los buses de datos se fija a 32 bit y se mantienen los mismos cinco canales básicos del protocolo completo. Al eliminar las ráfagas y las señales avanzadas, se reduce el consumo lógico de forma exponencial y la latencia es predecible, lo que se traduce en que por lo general se utiliza para leer y escribir registros de control en periféricos y aceleradores.

4.1.2.3 AXI4-Stream

Cuando los datos no tienen que ser direccionados (como es el caso de *streams* de video/audio/redes), la especificación *AXI4-Stream* elimina el campo de dirección y controla el flujo mediante un protocolo de *hand-shake* basado en señales TVALID/TREADY, junto con TDATA (datos), TLAST (fin de trama) y TUSER (marcadores de usuario) [27]. Esto posibilita el uso de una lógica mínima que es capaz de eliminar la sobrecarga y añadir *pipelines*, lo que permite confirmar que la especificación *AXI4-Stream* es la adecuada para la transferencia de flujos continuos de datos a muy altas velocidades.

4.2. FORMATO DE VÍDEO

En la plataforma desarrollada, se ha optado por elegir el formato YUV422 para extraer exclusivamente la información de luminancia (componente Y) asociada a cada píxel, imprescindible en la generación del mapa de profundidad que integra un procesamiento basado únicamente en intensidad.

En este esquema, la componente Y′ (o luma) procesa el brillo de la imagen a partir de la señal RGB, teniendo en cuenta la *corrección gamma* (transformación no lineal que se aplica a las señales de intensidad RGB antes de transmitirlas). Las señales de crominancia se representan con el símbolo U (también llamada Cb), que realiza la codificación de la diferencia entre el canal azul y la luminancia (B - Y′). Por otro lado, la componente V (o Cr) es la señal de crominancia que recoge la diferencia del componente rojo y su correspondiente luma (R - Y′). De este modo se separa de forma eficaz la señal de la intensidad de la señal del color y, además, se facilita el submuestreo.

Este formato se basa en que los seres humanos percibimos con mucha más agudeza los detalles de brillo que los de color, de modo que se puede reducir la resolución espacial de U y V sin pérdida visible de calidad. Este proceso es conocido como *chroma subsampling*.

4.2.1. Formato de píxel YUV 4:2:2

Para cada dos muestras de luma en una fila de píxeles, hay una muestra de U y una muestra V. Teniendo en cuenta que se usan 8 bits para codificar cada valor, se necesita un total de 4 bytes para cada dos píxeles (dos Y', una U y una V), para un promedio de 16 bits por píxel.

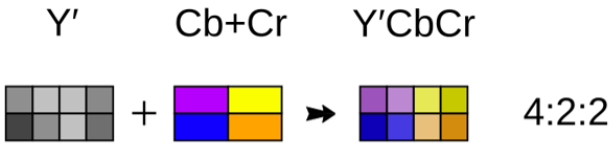

Figura 21. Submuestreo de crominancia 422 [28]

Por tanto, el muestreo de YUV 4:2:2 implica muestreo horizontal 2:1, sin muestreo vertical. Cada línea de un fotograma de la secuencia de vídeo, contiene cuatro muestras Y para cada dos muestras de U o V (Figura 21 y Figura 22).

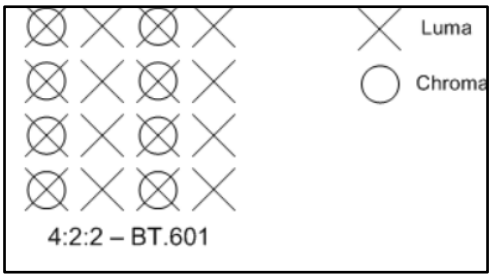

Figura 22. Muestreo YUV 422 [28]

En el formato YUYV, los datos se pueden tratar como una matriz de valores de caracteres sin signo, donde el primer byte contiene el primer byte Y, el segundo byte contiene la primera muestra U (Cb), el tercer byte contiene la segunda muestra Y y el cuarto byte contiene el primer byte V (Cr), como se refleja en el diagrama siguiente de la Figura 23.

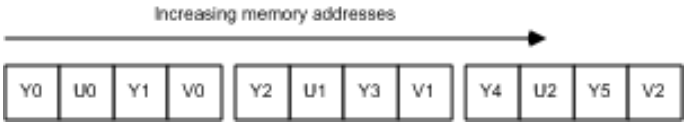

Figura 23. Avance de direcciones de memoria

Si la imagen se interpreta como una matriz de valores de palabra *little-endian* (16 bits), la primera palabra contiene la primera muestra Y en los bits menos significativos (LSB) y la primera muestra de U (Cb) en los bits más significativos (MSB). La segunda palabra de la primera muestra Y en los LSB, y la primera muestra V (Cr) en los MSB [29] .

4.2.2. Conversión Bayer a YUV y re-muestreo

El sensor OV5640 incluye un ISP que:

1. Realiza el demosaico del patrón Bayer GRBG, interpolando los valores de color de cada píxel para reconstruir la imagen en RGB.
2. Aplica la matriz BT.601 para generar vídeo YUV 4:2:2 (formato YUY2) antes de transmitirlo por la interfaz paralela.

Para su procesamiento en la sección PL del dispositivo ZYNQ integrado en la placa de prototipado PYNQ-Z2, se configuran los registros de salida para emitir los bytes en el orden $\{Y_0, Cb, Y_1, Cr\}$. De este modo, la PL recibe directamente los datos y puede, si se requiere,

extraer únicamente la componente de luminancia Y' utilizando la fórmula estándar (BT.601/709):

$$Y' = 0{,}299\,R + 0{,}587\,G + 0{,}114\,B \quad (1)$$

Como esta combinación captura casi toda la información de brillo que percibe el ojo humano, se pueden descartar las componentes de crominancia Cb/Cr sin afectar la calidad visual ni el rendimiento del sistema de visión estéreo propuesto, que solo utiliza niveles de luminancia.

Una vez se obtiene la señal de luminancia, se almacenará en formato YUY2 (YUV 4:2:2) con los bytes Cb y Cr siempre en 128 (valor "normal" según la norma Y'CbCr) para que opere con los IP de vídeo y con la conexión HDMI de la arquitectura HW de la plataforma. Así, se siguen enviando 16 bit por cada punto, pero solo los 8 bits correspondientes a la información de luminancia Y' tienen datos reales (equivalente al formato Y8). El uso de ancho de banda es un 33 % menor que con RGB y el color que falta no usa BRAM ni consume ciclos DMA. El resultado es una señal en blanco y negro en un formato común.

4.3. DESCRIPCIÓN DE LOS BLOQUES HARDWARE DE LA PLATAFORMA DE PRUEBA DEL ARRAY DE CÁMARAS

En este apartado se detallan los bloques IP empleados en el diseño de la arquitectura HW de la plataforma de visión estereoscópica inicial, que pretende mostrar la correcta implementación y funcionamiento del array de cámaras. Cada bloque cumple una función específica dentro de: la cadena de captura, almacenamiento y visualización de las imágenes. Se detalla desde la recepción de los datos crudos proporcionadas por cada cámara hasta su presentación en pantalla en escala de grises.

A la vista de la Figura 24, el diagrama de bloques plasma el recorrido completo que siguen los datos de vídeo desde los dos sensores OV5640 hasta la salida HDMI-DVI de la placa de prototipado PYNQ-Z2.

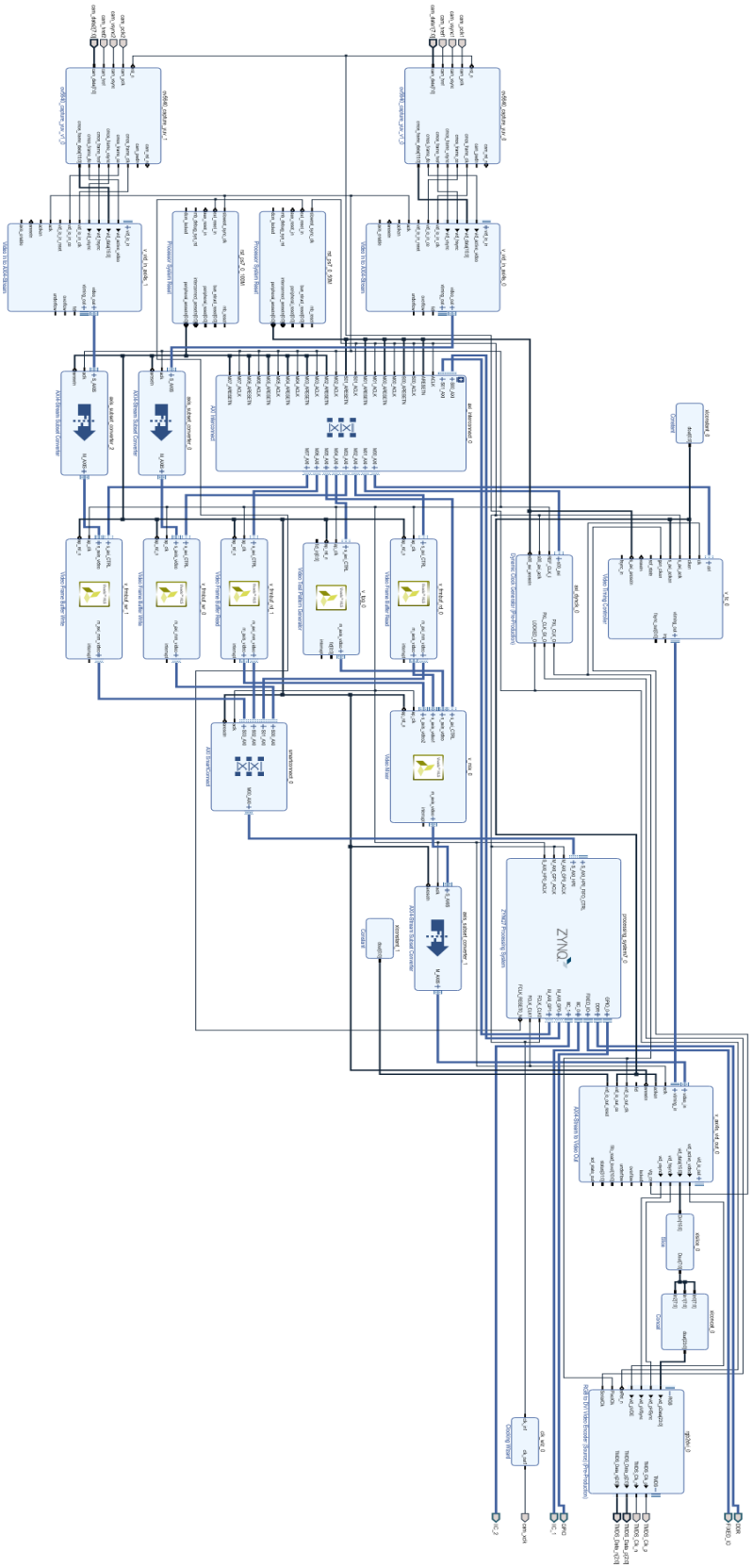

Figura 24. Diagrama de bloques de la arquitectura hardware del array de cámaras YUV422

Los módulos de captura a medida denominados, *ov5640_capture_yuv_0/1*, se localizan en la parte izquierda/superior y son los responsables de muestrear el bus DVP de cada uno de los sensores OV5640 del array de cámaras, así como de sincronizar las señales VSYNC/HREF y de agrupar los bytes YUYV en palabras de 16 bits. Cada flujo de vídeo paralelo, junto con su señal de reloj PCLK, se dirige al bloque *Video In to AXI4-Stream* proporcionado en el catálogo de IP de AMD/Xilinx. Este bloque convierte la señal del dominio del sensor al formato *AXI4-Stream* dentro del PL y añade los metadatos de resolución y sincronismo necesarios.

Una vez serializados, los píxeles son escritos en memoria DDR a través de los IP *Video Frame Buffer Write* asociados a cada uno de los sensores. El procesador Zynq (PS) tiene acceso a estas regiones gracias al puerto de alto rendimiento S_AXI_HP0. A su vez, los IP *Video Frame Buffer Read* permiten que se recuperen los *frames* almacenados en tiempo real. Las dos salidas *AXI-Stream* confluyen en el IP *Video Mixer*, que permite generar una composición lado a lado y poder ver las dos imágenes simultáneamente.

La temporización global se realiza bajo el control del núcleo *Video Timing Controller* (VTC), cuyo reloj de píxel está proporcionado por el IP *Dynamic Clock Generator*. El flujo mezclado se encuentra en un convertidor *AXI4-Stream Subset Converter*, quien lo convierte a continuación en un *AXI4-Stream to Video Out*, donde se vuelven a insertar las señales HSYNC/VSYNC junto con la ventana activa. Un pequeño módulo de tipo *Slice/Concat* mantiene el byte de luminancia Y, replicándolo en R-G-B para mostrarse en ausencia de crominancia. Finalmente, el núcleo *RGB to DVI Encoder* envía 24 bits RGB los cuales se codifican en pares diferenciales TMDS para el conector HDMI. Así, se concluye con la ruta de captura-procesado-visualización.

En este diseño el dispositivo APSoC *Zynq-7000* juega un doble rol: inicializa sensores e IPs a través de su interfaz *AXI-Lite (M_AXI_GP0/1)*, mientras gestiona los buffers vía DMA de alto rendimiento. Todo el subsistema se sincroniza a través de la generación de un reloj *FCLK_CLK0* con una frecuencia de 50 MHz para lógica AXI y una señal de reloj *FCLK_CLK1* de 100 MHz para vídeo, y su configuración.

Con el fin de ofrecer una descripción exhaustiva del sistema, en las secciones 4.3.1 a 4.3.13, se explica en detalle el propósito de cada uno de los IPs del diagrama de bloques de la plataforma.

4.3.1. IP de captura ad-hoc de datos desde el sensor OV5640

El bloque IP *ad-hoc ov5640_capture_yuv* de la Figura 25 (implementado en lenguaje VHDL) se encarga de capturar el flujo de datos de un sensor de cámara OV5640 a través de su interfaz paralela DVP. Implementa la lógica de sincronización necesaria para interpretar correctamente las señales de sincronismo provenientes del sensor: en particular la señal VSYNC (*frame sync*) de la cámara, que indica un cuadro de imagen válido activo (nivel alto durante la duración del *frame*), y la señal HREF (también denominada HSYNC), que se encuentra a nivel alto durante cada línea activa de píxeles.

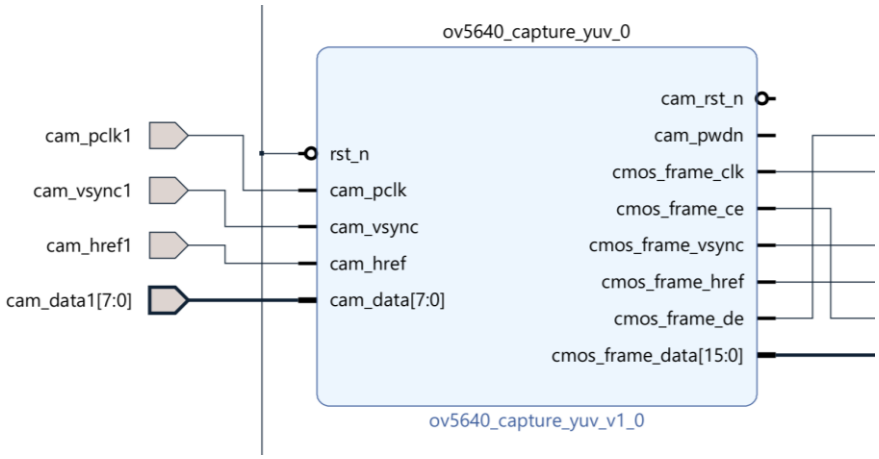

Figura 25. IP de la cámara OV5640

En cada ciclo del reloj de píxel de la cámara (señal PCLK), el bloque toma el dato de 8 bits del bus DOUT[7:0] del sensor OV5640 correspondiente a un componente de color (ya configurado el sensor para emitir en formato YUV 4:2:2). La lógica interna agrupa estos *bytes* en palabras de 16 bits siguiendo el empaquetado YUYV propio del formato YUV 4:2:2, donde los valores de crominancia U y V se comparten entre píxeles adyacentes. Concretamente, el bloque combina cada par de bytes consecutivos en la secuencia YUV de la cámara para formar palabras de 16 bits en formato YUYV como se muestra en el Código 1.

Código 1. Agrupación de bytes en palabras de 16 bits

```
if cam_href = '1' then
  byte_flag <= not byte_flag;
  cam_data_d0 <= cam_data;
  if byte_flag = '1' then
    cmos_data_16b <= cam_data & cam_data_d0;
  end if;
end if;
```

Donde:

```
cam_data              : in  std_logic_vector(7 downto 0); -- Datos de la cámara
signal cam_data_d0    : std_logic_vector(7 downto 0) := (others => '0');
signal cmos_data_16b  : std_logic_vector(15 downto 0) := (others => '0');
```

Con el fin de alinear y sincronizar la salida, este IP utiliza las señales VSYNC/HREF de entrada para habilitar únicamente los datos válidos. Durante los periodos en que VSYNC está activo (*frame* en curso) y HREF indica píxeles válidos en una línea, el bloque activa una señal interna de habilitación de muestreo de píxel. Esta señal de *clock enable* de fotograma, denominada *frame_ce*, se establece a nivel alto en los ciclos de reloj en los que la salida contiene datos de píxel válidos, como se muestra en el Código 2. En otras palabras, *frame_ce* actúa como señal de *validación de datos* coincidente con cada palabra YUYV de 16 bits proporcionado en la salida correspondiente del IP, indicando a la etapa siguiente que en ese ciclo hay un píxel válido en el bus de datos.

Código 2. Generación de señales de habilitación y sincronización

```
    internal_cmos_frame_ce <= (wait_done and (byte_flag_d0 and internal_cmos_frame_href)) or
(not internal_cmos_frame_href);
    cmos_frame_vsync <= wait_done and cam_vsync_d1;
    internal_cmos_frame_href  <= wait_done and cam_href_d1;
    internal_cmos_frame_de  <= internal_cmos_frame_href;
    cmos_frame_data <= cmos_data_16b when (wait_done = '1') else (others => '0');

    --Asignación de las señales internas a la salida
    cmos_frame_ce <= internal_cmos_frame_ce;
    cmos_frame_href <= internal_cmos_frame_href;
    cmos_frame_de <= internal_cmos_frame_de;
```

4.3.2. IP *Video In to AXI4-Stream*

El IP *Video In to AXI4-Stream* proporcionado por AMD/Xilinx en el catálogo de IPs del entorno Vivado 2023.1 y cuya interfaz E/S se representa en la Figura 26, actúa como pasarela entre la interfaz paralela procedente del sensor y el *backbone AXI4-Stream* del dispositivo Zynq.

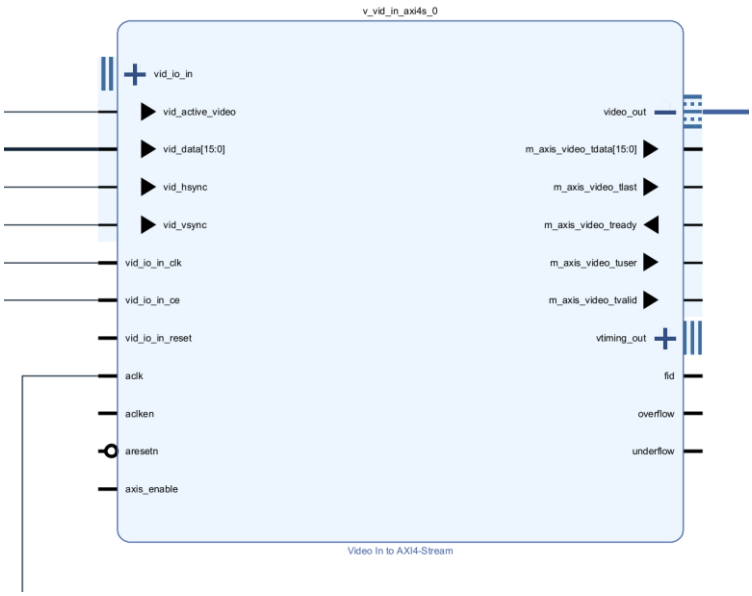

Figura 26. IP Video In to AXI4-Stream configurado en formato YUV 422

Recibe los datos YUYV de 16 bits (YUV 4:2:2) junto con las señales VSYNC y HREF generadas por el bloque de captura de datos de la cámara OV5640 y, sin alterar el contenido de los píxeles, los encapsula en un flujo AXI: cada beat se coloca en el bus *tdata*, se marca el primer píxel del cuadro con *tuser*=1 (*Start of Frame*) y se indica el final de cada línea con *tlast*=1. De este modo, toda la información de sincronismo queda embebida en el propio protocolo *AXI4-Stream*, suprimiendo la necesidad de propagar las líneas discretas HSYNC/VSYNC. El núcleo respeta el ancho de palabra de 16 bits, mantiene la secuencia Y-U-Y-V original y se limita a convertir la forma de señal, lo que posibilita conectar directamente con el IP *Frame Buffer Write/Read*, mezcladores de vídeo o aceleradores sin lógica adicional.

4.3.3. IP *AXI4-Stream Subset Converter*

El *AXI4-Stream Subset Converter* es un bloque de infraestructura diseñado para adaptar y armonizar diferencias menores entre interfaces *AXI4-Stream* de distintos módulos. Permite

cambiar las características de la interfaz *AXI4-Stream* entre un dispositivo maestro y un dispositivo esclavo, por ejemplo, ajustando el ancho de bus de datos o eliminando/agregando señales de control para que ambas partes sean compatibles (Figura 27).

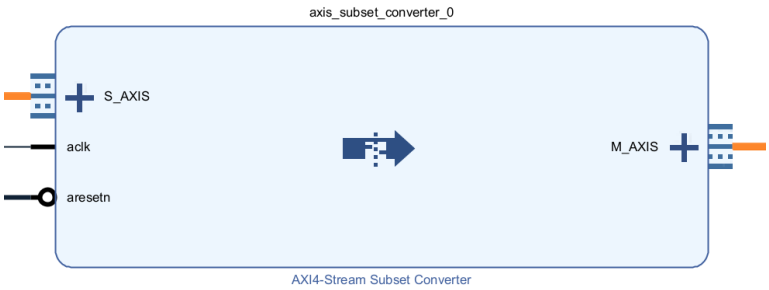

Figura 27. IP AXI4-Stream Subset Converter

En el diseño de la arquitectura HW inicial de la plataforma, este IP asegura que el tamaño de los datos del flujo *AXI4-Stream* de vídeo generado tras la captura, coincida con el ancho de bus de la siguiente etapa, en este caso, los núcleos *Frame Buffers* de memoria. Así no hay necesidad de modificar manualmente ninguno de los bloques principales.

El núcleo IP *AXI4-Stream Subset Converter* se configura definiendo un ancho de datos (TDATA *Witdh*), en bytes, la interfaz de entrada (*Slave Interface*) y otro para la de salida (*Master Interface*) que coincidan con los que necesitan los bloques adyacentes[30].

En la Figura 28 se presenta la configuración del IP *AXI4-Stream Subset Converter* para ampliar el ancho de datos de 2 a 3 bytes. En este esquema, la salida del bloque *Video In to AXI4-Stream* permanece en un bus de 16 bits con formato YUYV, mientras que la entrada a los *frame buffers* está definida por defecto como un bus de 24 bits, independientemente del formato de vídeo. Para homogeneizar ambos anchos, los 16 bit originales se asignan a `tdata[15:0]` y los 8 bits más significativos (`tdata[23:16]`) se rellenan con ceros (`8'b00000000`), obteniéndose así un flujo de 24 bits compatible con el *buffer*.

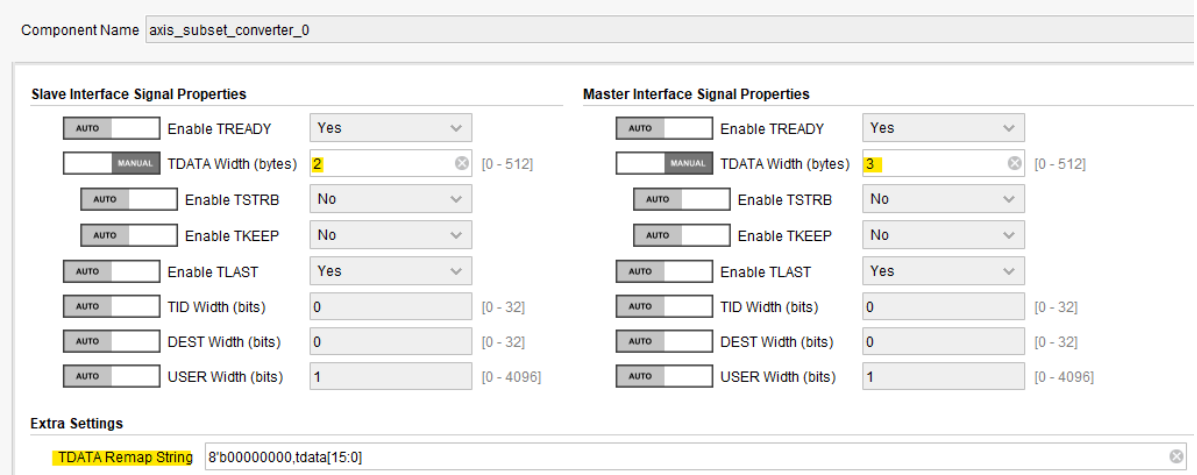

Figura 28. Configuración del AXI4-Stream Subset Converter

Este IP es una pasarela de compatibilidad que no modifica el contenido de la imagen ni realiza ningún tipo de procesado, utilizándose para evitar advertencias críticas *(critical warnings*) de incompatibilidad de ancho de bus en el diseño Vivado. Se seguirá utilizando a lo largo del flujo del diseño para evitar disparidades aumentando o reduciendo los anchos de buses.

4.3.4. IP *Video Frame Buffer Write*

Los bloques *Frame Buffer* están diseñados para aplicaciones de vídeo que requieren *buffers* de fotogramas y accesos a altos anchos de banda entre la interfaz de vídeo *AXI4-Stream* y la interfaz AXI4 [31]. En concreto, el IP *Video Frame Buffer Write* (Figura 29), es el complemento de escritura en memoria, que recibe un flujo de vídeo *AXI4-Stream* y lo almacena en memoria DDR como un *frame* rasterizado.

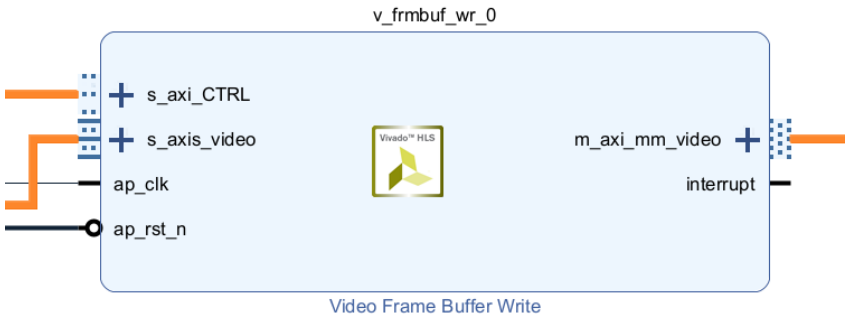

Figura 29. IP Video Frame Buffer Write

En el diseño de la arquitectura HW de la plataforma inicial, el IP *Video Frame Buffer Write* toma el *stream* proveniente de la cámara, ya transportado por el IP *Video In to AXI4-Stream* y adaptado por el IP *AXI4-Stream Subset Converter*, y lo almacena en la memoria DDR integrada en la placa de prototipado PYNQ-Z2 en formato YUYV8 (Figura 30). Por tanto, por cada 2 píxeles capturados, el núcleo escribe 4 bytes en memoria en orden YUYV.

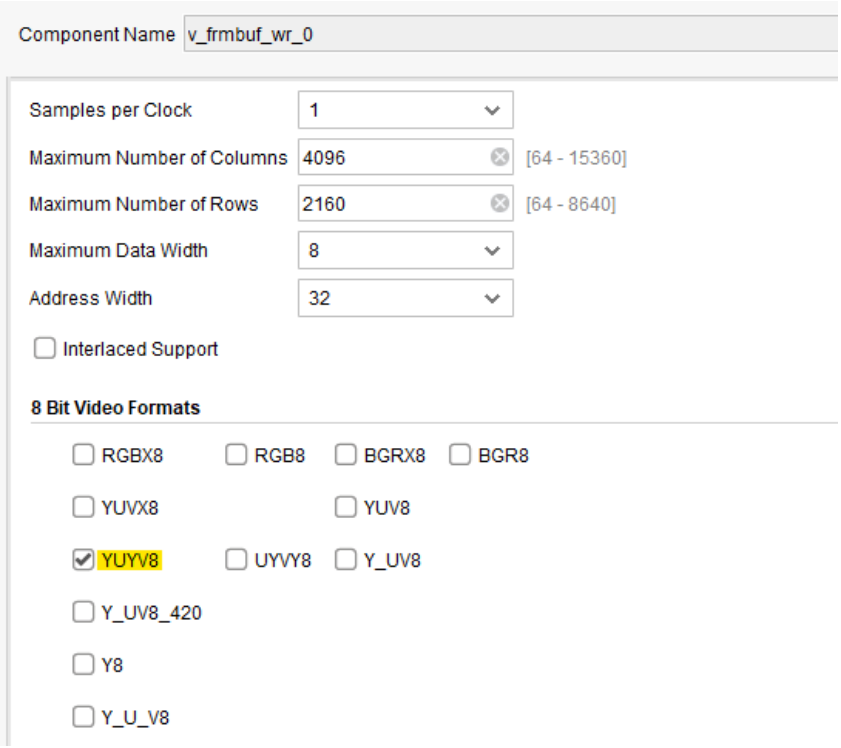

Figura 30. Configuración del IP Video Frame Buffer Write

El uso de este IP es fundamental, puesto que actúa como *buffer* de captura. Almacena cada cuadro para que pueda ser usado posteriormente, permitiendo implementar un doble *buffer* para evitar *tearing* o la posibilidad de que el procesador acceda a los *frames* en memoria sin interferir con la captura en curso.

4.3.5. IP *Video Frame Buffer Read*

Complementario al IP *Video Frame Buffer Write*, la función del IP *Video Frame Buffer Read* (Figura 31) es leer los *frames* de vídeo previamente almacenados en memoria DDR y enviarlos nuevamente en formato *AXI4-Stream* hacia los módulos de salida.

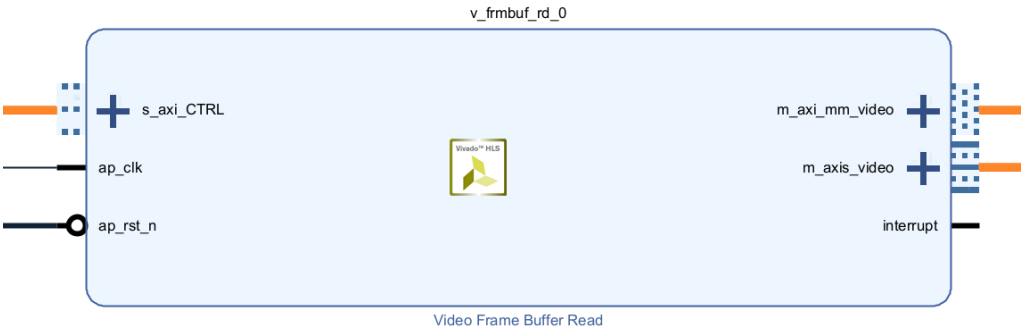

Figura 31. IP Video Frame Buffer Read

En la arquitectura HW de la plataforma inicial, este IP se configura para leer los *frames* en el mismo formato YUYV8 (Figura 32). El IP gestiona las solicitudes AXI al controlador de memoria *m_axi_mm_video* para recuperar los datos línea a línea. A medida que extrae píxeles de la memoria DDR, el IP los convierte en un flujo *AXI4-Stream* de salida (*m_axis_video*), reproduciendo la sincronización de vídeo original. El primer píxel del *frame* se marca con *tuser*=1 al inicio del *frame* y se genera *tlast*=1 al final de cada fila de píxeles. De esta forma se asegura que el flujo de salida vuelva a estar correctamente delimitado en líneas y cuadros.

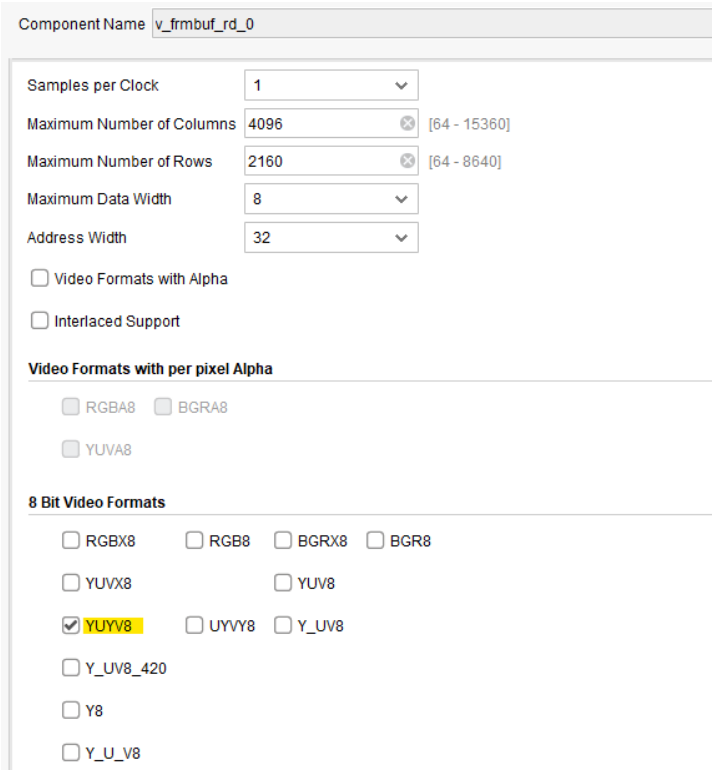

Figura 32. Configuración del IP Video Frame Buffer Read

La elección de usar los IP *Video Frame Buffers* en lugar del núcleo VDMA (*Video Direct Memory Acess*) que escribe y lee en memoria en modo triple buffer, se debe a una recomendación de AMD/Xilinx, que valora la sustitución de este IP por los *buffers Write* y *Read* para una mejor gestión de los recursos *hardware*.

4.3.6. IP *Video Test Pattern Generator*

El IP *Video Test Pattern Generator* (TPG) de la Figura 33 es un bloque IP que genera secuencias de vídeo sintéticas con patrones estándar como barras de color, escalas de grises, cuadros, etc. Es de gran utilidad para probar y calibrar la cadena de procesamiento de vídeo sin necesidad de una fuente externa [32]. En el flujo implementado, el núcleo TPG se utiliza como fuente inicial de vídeo para verificar la salida HDMI, lo que permite validar la estabilidad del sistema a distintas resoluciones. Genera una salida *AXI4-Stream* que se proporciona al bloque posterior *Video Mixer* como *layer master*, que marca la señal de entrada de resolución definida permitiendo la superposición de vídeo de las cámaras izquierda y derecha en la región definida de la pantalla.

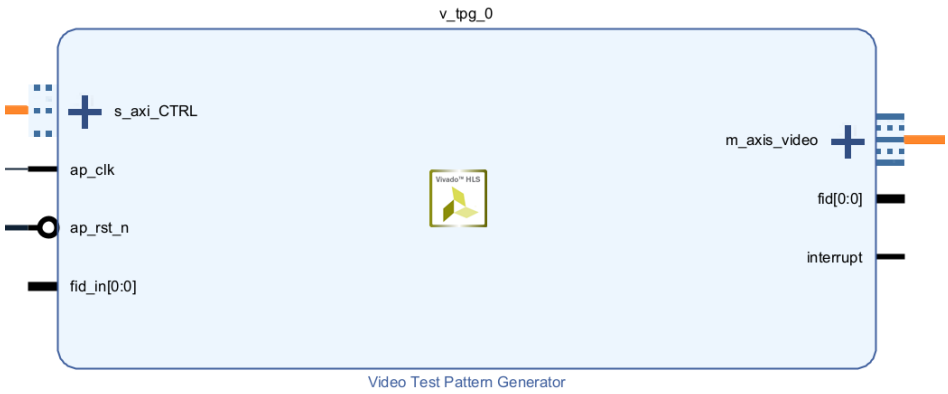

Figura 33. IP Video Test Pattern Generator

La Figura 34 muestra la configuración del bloque IP *Test Pattern Generator* (TPG) con soporte YUV 4:2:2 compatible con *AXI4-Stream*, y los distintos patrones de prueba que puede generar (como, rampa de luminosidad, barras de colores, cuadro de retícula, entre otros).

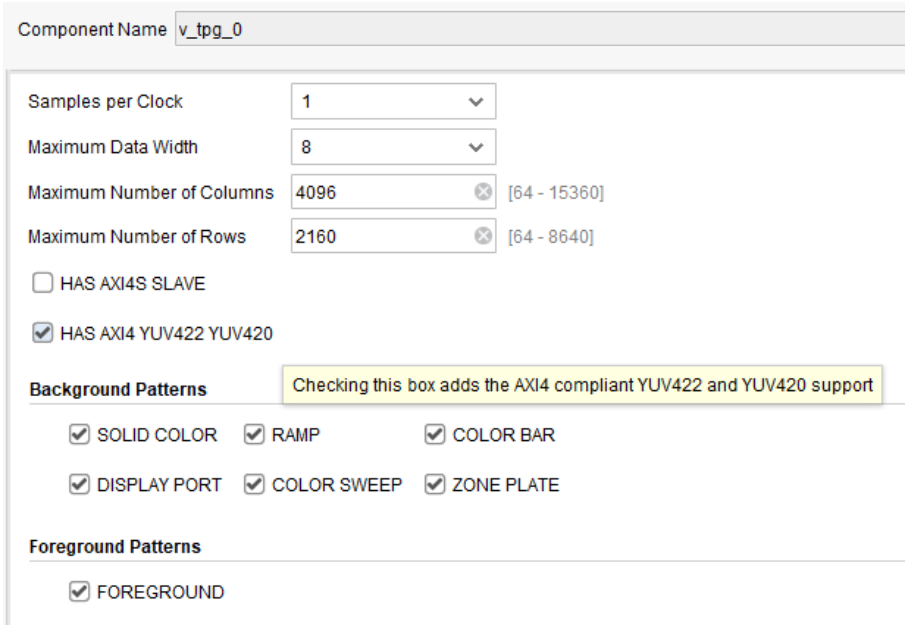

Figura 34. Configuración del generador de patrones de vídeo

4.3.7. IP *Video Mixer*

El *Video Mixer* (Figura 35) es un IP de mezcla y composición de capas de vídeo. Su funcionalidad dentro del flujo implementado consiste en combinar múltiples flujos provenientes de las dos cámaras estéreo y el TPG en un solo flujo de salida de tipo *AXI4-Stream*. En el diseño de la arquitectura HW de la plataforma inical, el este IP se configura para permitir la visualización simultánea de la imagen izquierda y derecha en un mismo *frame,* colocándolas lado a lado superponiéndose sobre la imagen del TPG de fondo.

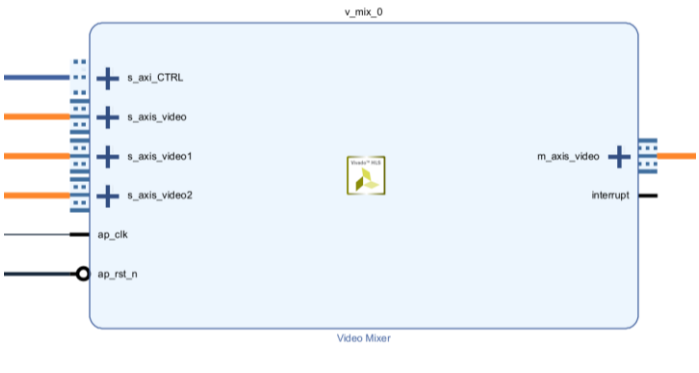

Figura 35. IP Video Mixer

Este bloque está configurado para 2 capas de vídeo más una adicional que actúa como *layer master*. Como se muestra en la Figura 36, cada capa puede tener su propia resolución, ajustándose en este caso al máximo permitido de ancho de *buffer*, que tiene que ser la mitad del máximo número de columnas soportadas por el IP *Video Mixer*. Además, el IP ofrece funciones integradas de escalado y *alpha blending* por capa, eliminando la necesidad de bloques adicionales dedicados a estas tareas.

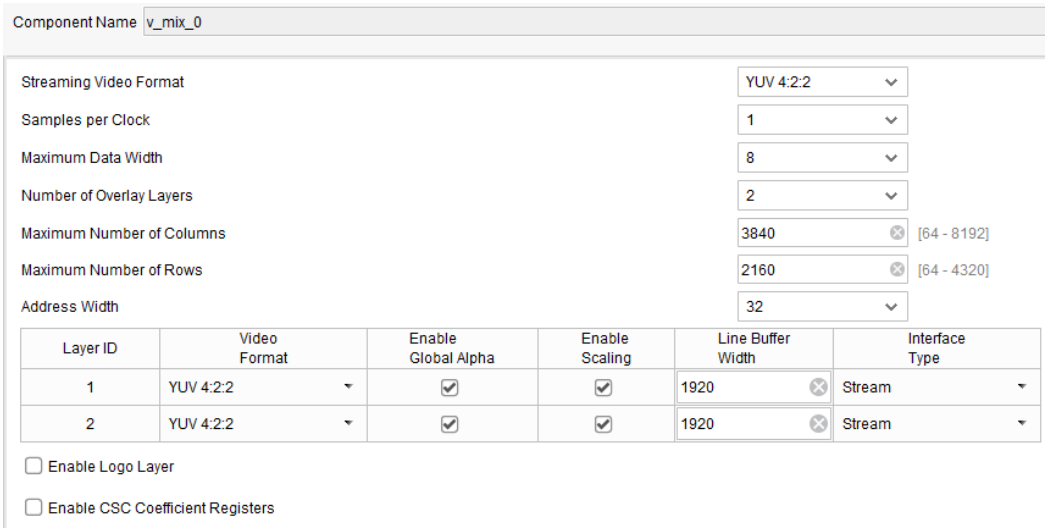

Figura 36. Configuración del Video Mixer

4.3.8. IP *Video Timing Controller*

El IP *Video Timing Controller* (VTC) de la Figura 37 es un detector y generador de temporización de vídeo de uso general que detecta automáticamente la supresión de señal y la temporización de datos activos basándose en los pulsos de sincronización horizontal y vertical de entrada. Este núcleo se utiliza en conjunto con el núcleo IP *AXI4-Stream to Video Out* para detectar el formato y la temporización del vídeo entrante [33].

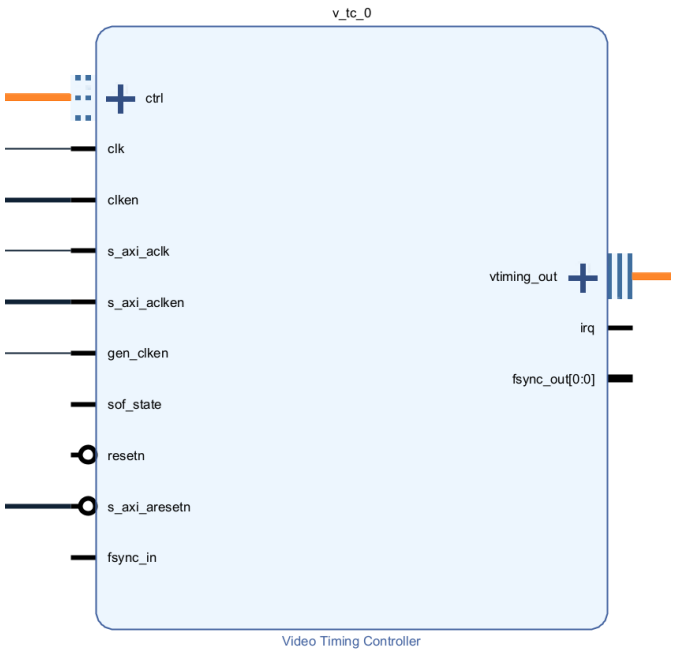

Figura 37. IP Video Timing Controller

En la arquitectura HW de la plataforma inicial, el IP VTC se utiliza para generar los distintos modos de vídeo, cargando la configuración correspondiente a cada estándar vía interfaz *AXI4-Lite*. Como se muestra en las opciones de configuración de la Figura 38, se habilita dicha interfaz para permitir la reprogramación en tiempo real de la temporización desde software, de manera que el sistema puede cambiar la resolución sin necesidad de re-sintetizar el diseño.

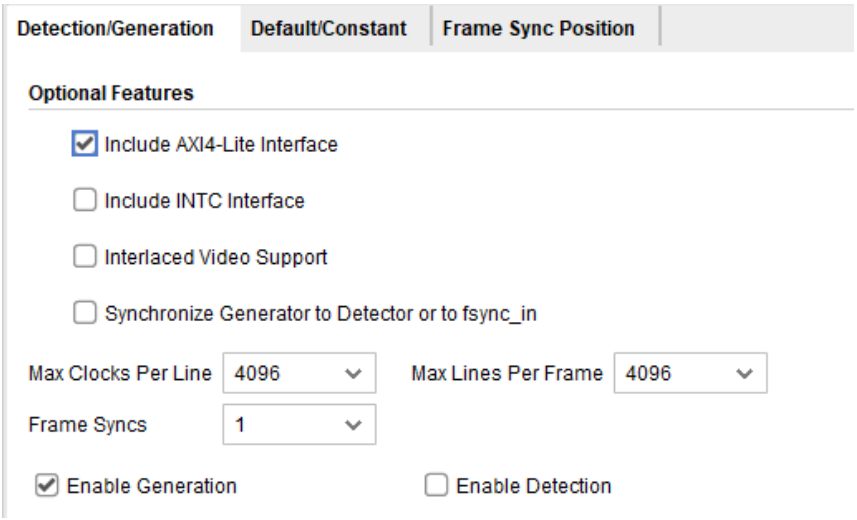

Figura 38. Configuración de generación de timing

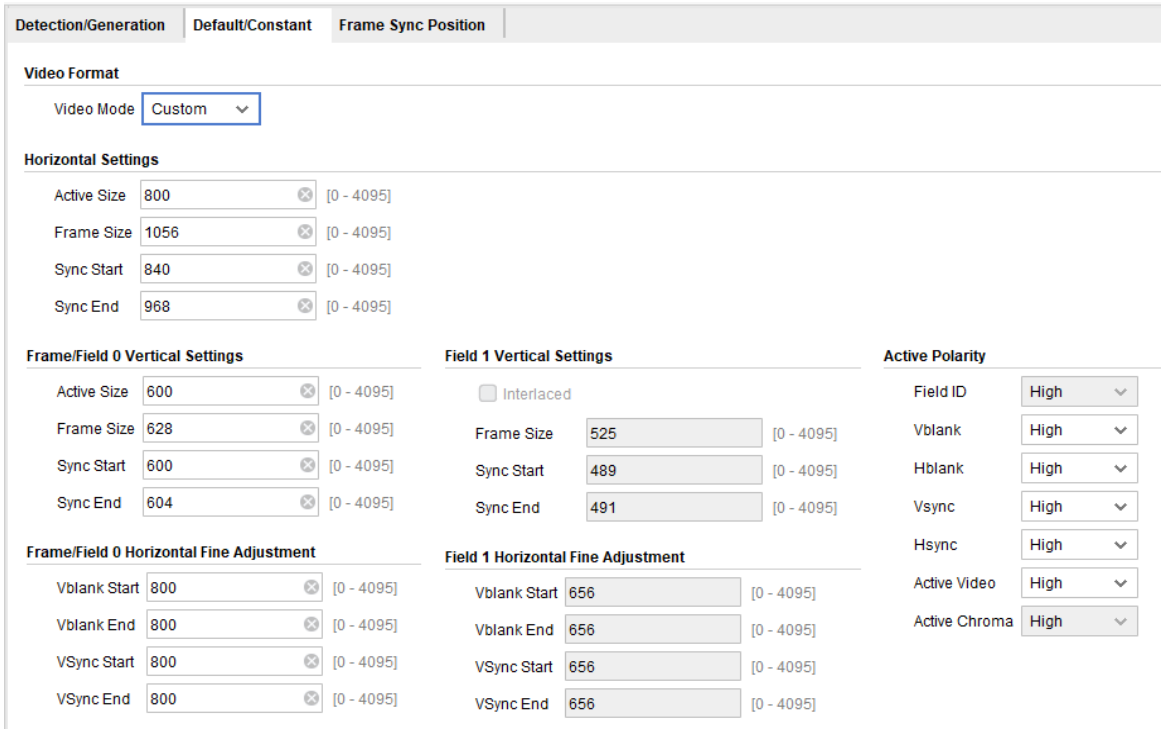

Figura 39. Configuraciones de frame y campo

El IP VTC recibe como reloj de referencia el reloj de píxel generado por el IP *Dynamic Clock Generator* (descrito en la siguiente sección). Sus salidas de sincronismo se conectan al IP *AXI4-Stream to Video Out*, actuando como maestro de sincronización. Éste genera continuamente la cadencia de sincronismos según los parámetros cargados (Figura 39). Estas señales indican el timing exacto de cuándo comienza y termina cada línea y cuadro, siguiendo el estándar VESA-DMT (Figura 40). Así se garantiza que el IP *AXI4 Stream to Video Out* disponga de toda la información de temporización.

Detailed Timing Parameters

Timing Name	= **800 x 600 @ 60Hz;**							
Hor Pixels	= **800;**	// Pixels						
Ver Pixels	= **600;**	// Lines						
Hor Frequency	= 37.879;	// kHz	=	26.4	usec	/	line	
Ver Frequency	= 60.317;	// Hz	=	16.6	msec	/	frame	
Pixel Clock	= **40.000;**	// MHz	=	25.0	nsec	± 0.5%		
Character Width	= **8;**	// Pixels	=	200.0	nsec			
Scan Type	= **NONINTERLACED;**			// H Phase	=	2.3 %		
Hor Sync Polarity	= **POSITIVE;**	// HBlank	=	24.2%	of HTotal			
Ver Sync Polarity	= **POSITIVE;**	// VBlank	=	4.5%	of VTotal			
Hor Total Time	= 26.400;	// (usec)	=	132	chars	=	1056	Pixels
Hor Addr Time	= 20.000;	// (usec)	=	100	chars	=	800	Pixels
Hor Blank Start	= 20.000;	// (usec)	=	100	chars	=	800	Pixels
Hor Blank Time	= 6.400;	// (usec)	=	32	chars	=	256	Pixels
Hor Sync Start	= 21.000;	// (usec)	=	105	chars	=	840	Pixels
// H Right Border	= 0.000;	// (usec)	=	**0**	chars	=	0	Pixels
// H Front Porch	= 1.000;	// (usec)	=	**5**	chars	=	40	Pixels
Hor Sync Time	= 3.200;	// (usec)	=	**16**	chars	=	128	Pixels
// H Back Porch	= 2.200;	// (usec)	=	**11**	chars	=	88	Pixels
// H Left Border	= 0.000;	// (usec)	=	**0**	chars	=	0	Pixels
Ver Total Time	= 16.579;	// (msec)	=	628	lines	HT – (1.06xHA)		
Ver Addr Time	= 15.840;	// (msec)	=	600	lines	= 5.2		
Ver Blank Start	= 15.840;	// (msec)	=	600	lines			
Ver Blank Time	= 0.739;	// (msec)	=	28	lines			
Ver Sync Start	= 15.866;	// (msec)	=	601	lines			
// V Bottom Border	= 0.000;	// (msec)	=	**0**	lines			
// V Front Porch	= 0.026;	// (msec)	=	**1**	lines			
Ver Sync Time	= 0.106;	// (msec)	=	**4**	lines			
// V Back Porch	= 0.607;	// (msec)	=	**23**	lines			
// V Top Border	= 0.000;	// (msec)	=	**0**	lines			

Figura 40. Tabla de parámetros de sincronización de video para 800 x 600 @ 60 Hz

4.3.9. IP *Dynamic Clock Generator*

La salida de video HDMI requiere de una frecuencia de reloj de píxel concreta para cada resolución. Como marca el estándar VESA-DMT mencionado en el apartado anterior, para una señal 800x600 a 60 Hz se requiere un reloj de píxel de 40 MHz, mientras que para una señal de 1280x720 a 60 Hz la frecuencia de la señal de reloj de píxel debe ser 74.25 MHz.

Para proporcionar estas frecuencias, se incorpora en el diseño el IP *Dynamic Clock Generator* (Figura 41), que Digilent distribuye en sus diseños HDMI para placas Zynq-7000. Este IP toma como referencia el reloj de 100 MHz procedente de la sección PS del dispositivo ZYNQ-7000 (FCLK1), y genera:

- Un reloj de píxel con frecuencia programable (*PXL_CLK_0*)

- Un reloj 5x píxel que requieren los serlializadores TMDS del IP *RGB to DVI Video Encoder* (mencionado en el apartado 4.3.11. IP *RGB to DVI Video Encoder*)
- Un indicador de *lock* del MMCM interno

Estos parámetros se configuran a través de la interfaz AXI-Lite (s00_axi), que permite al procesador calcular y cargar dinámicamente los coeficientes multiplicador (M), divisor (D) y salida (O) del MMCM del núcleo.

Figura 41. IP Dynamic Clock Generator

4.3.10. IP *AXI4-Stream to Video Out*

El módulo *AXI4-Stream to Video Out* (Figura 42) es el responsable de transformar el flujo de datos de vídeo en formato *AXI4-Stream* (el que se usa internamente en los núcleos de procesamiento) a las señales de vídeo en paralelo con sincronismos separados, listas para enviarse a un monitor externo. Este núcleo funciona como una interfaz de salida que recibe los píxeles en forma de ráfaga (con *tvalid, tready, tlast, tuser,* etc.) y los pone en sincronismo con las señales de *timing* del VTC para generar una salida continua.

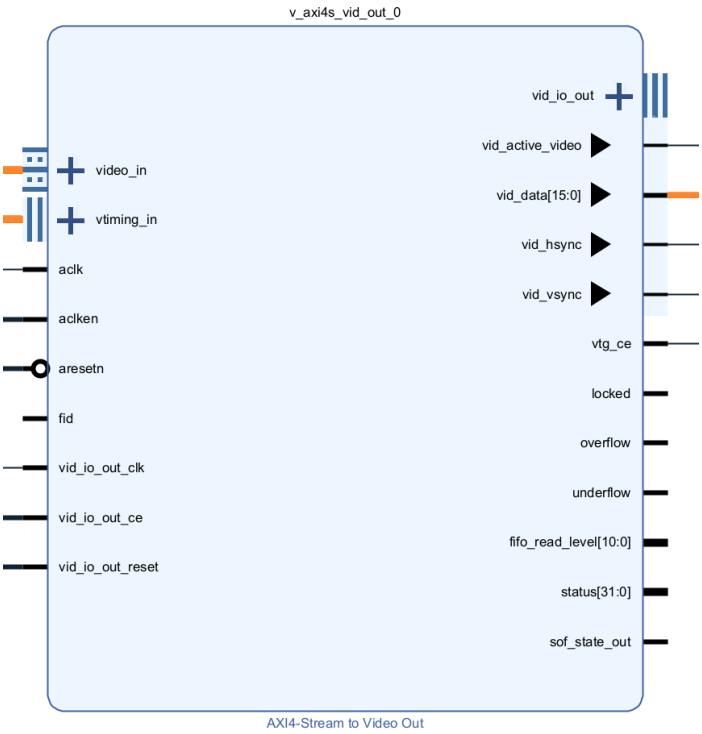

Figura 42. IP AXI4-Stream to Video Out

En la configuración del diseño propuesto, la salida del IP *Video Mixer* se conecta al módulo *AXI4-Stream Subset Converter* que, a su vez, se conecta a la entrada *video_in* del IP *AXI4-Stream to Video Out* configurado para procesar píxeles en formato YUV422 (Figura 43).

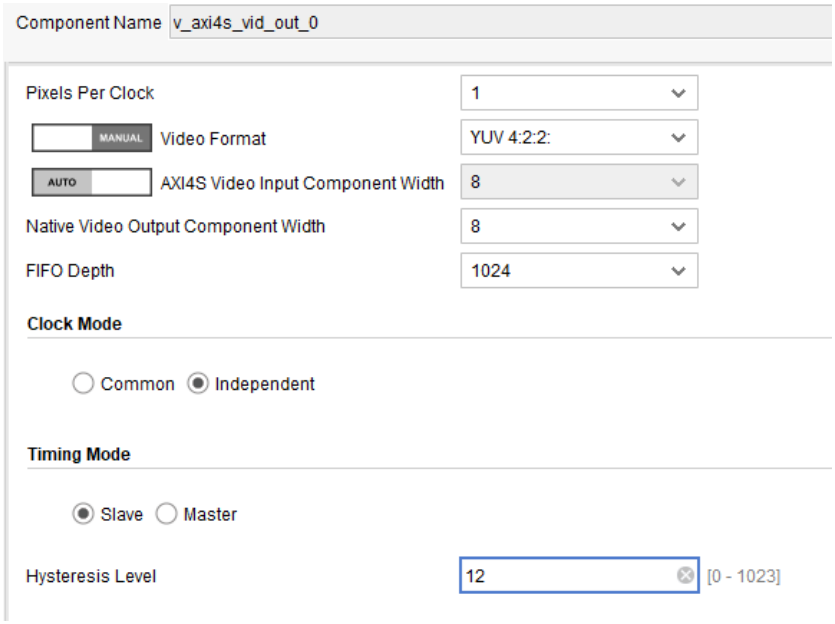

Figura 43. Configuración del IP AXI4-Stream to Video Out

La señal de salida *vid_data* generada por este IP se lleva a la entrada del módulo codificador *RGB to DVI* (mostrado en el apartado 4.3.11. IP *RGB to DVI Video Encoder*). Previamente, el flujo de datos se envía a unos bloques de tipo *Slice* y *Concat* (Figura 44), ya que solo se necesitan los 8 bits LSB asociados a la componente de luma (Y) para generar la imagen en escala de grises. Posteriormente, el bloque *Concat* replica ese byte Y en los tres octetos que forman el bus de 24 bits requerido por el IP *RGB to DVI Video Encoder*.

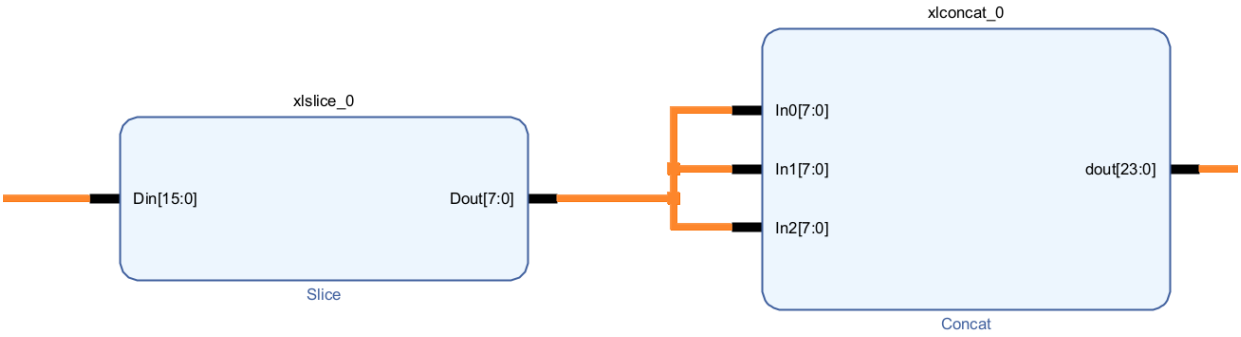

Figura 44. Separación y recomposición de datos mediante Slice y Concat

4.3.11. IP *RGB to DVI Video Encoder*

La última etapa de la cadena de vídeo consiste en convertir la señal RGB de 24 bits en los pares diferenciales TMDS requeridos por la salida HDMI-DVI de la placa de prototipado PYNQ-Z2. Para ello se emplea el núcleo *RGB to DVI Video Encoder* de Digilent, mostrado en la Figura 45.

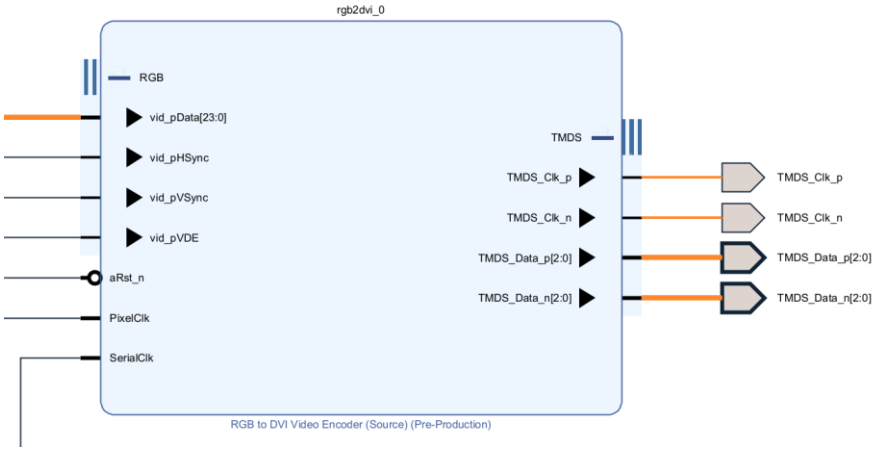

Figura 45. IP RGB to DVI Video Encoder

Este IP reúne cada píxel de 24 bits: 8 bits R, 8 bits G, 8 bits B (en este caso [Y, Y, Y]), además de las señales de sincronización (*pHSync*, *pVSync*) y activación de vídeo (*pVDE*), en símbolos TMDS de 10 bits. Tres flujos de datos, junto con uno de reloj, se serializan a diez veces la velocidad de *PixelClk*. Esto da como resultado los pares diferenciales *TMDS Data[2:0] P/N* y *TMDS Clk P/N*, que se enlazan sin intermediarios al puerto HDMI de la tarjeta[34] (Figura 46). En nuestra configuración, para la serialización, se precisa de un reloj *SerialClk* generado por el múdulo *Dynamic Clock Generator* cuya frecuencia cinco veces superior al de *PixelClk* (*PXL_CLK_5X_O*).

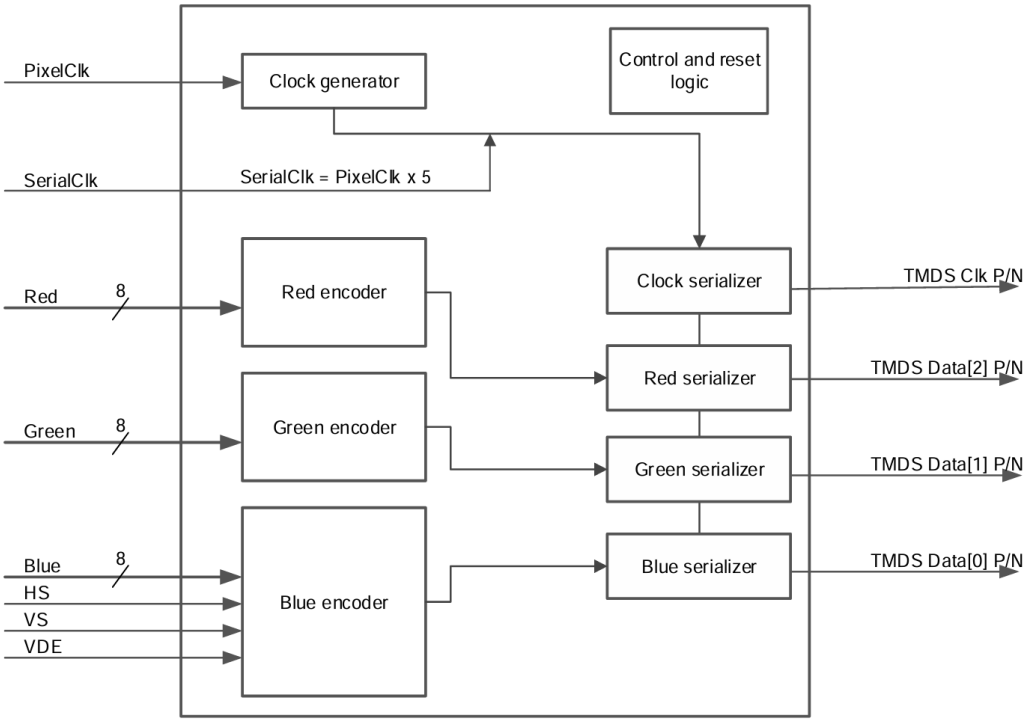

Figura 46.Diagrama de bloques RGB to DVI converter

Finalmente, los pares TMDS se enrutan hacia los pines diferenciales del puerto HDMI que se definen en el archivo de restricciones XDC del proyecto (4.4. Archivo de restricciones).

Esta fase pone fin a la secuencia completa de captura, procesamiento y despliegue de la arquitectura HW de la plataforma inicial, posibilitando el control continuo de la imagen monocromática mediante un monitor DVI/HDMI común.

4.3.12. IP *ZYNQ-7000 Processing System*

El bloque ZYNQ7 *Processing System*, representa la sección de procesamiento (PS) del APSoC XC7Z020, integrado en la placa de prototipado PYNQ-Z2. Su configuración representa el inicio del diseño, ya que, entre otras funciones, muestra el controlador de la DDR, el esquema de reloj y las conexiones AXI que facilitan que la lógica programable (PL) se conecte con la memoria y los componentes del sistema, así como los controladores I2C que permitirán configurar los registros de los sensores OV5640 de cada uno de los arrays. La visión global de este núcleo se aprecia en la Figura 47.

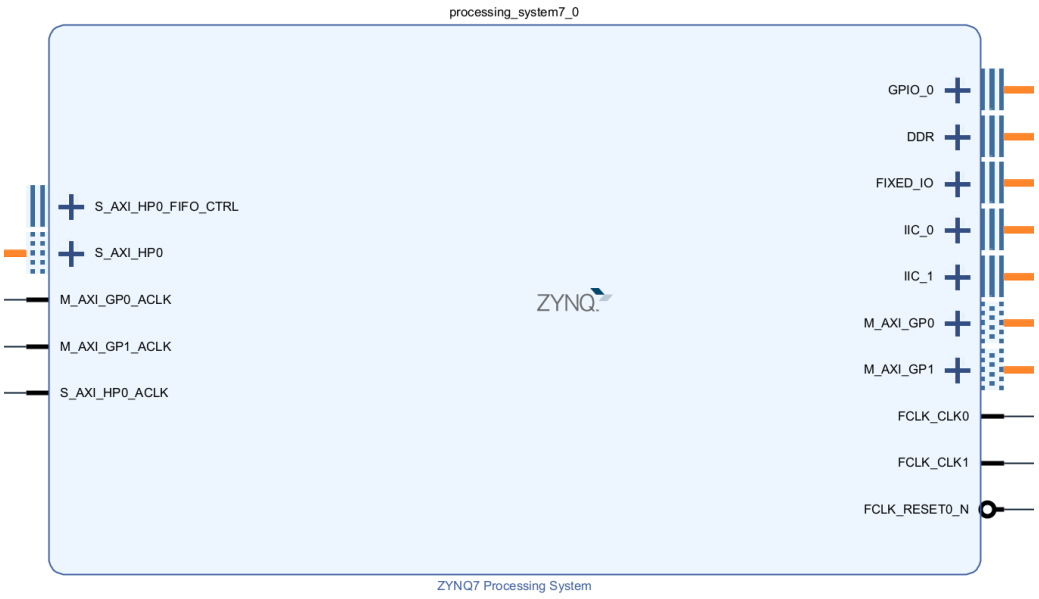

Figura 47. IP ZYNQ7 Processing System

En la sección de Configuración PS-PL (Figura 48), se activan las siguientes interfaces AXI para dar soporte a los requerimientos de la plataforma inicial:

- M AXI GP0 y M AXI GP1 (*General Purpose AXI Master Interface*): Una interfaz simplificada (AXI-Lite) que la aplicación sin sistema operativo utiliza para gestionar los registros de control de los componentes IP de la arquitectura HW.
- S AXI HP0 (*AXI High Performance Slave Interface*): Vías rápidas que enlazan los IP *Video Frame Buffers* con la memoria DDR, asegurando un ancho de banda estable superior a 1 GiB/s.

Este ajuste simplifica que el procesador gestione la recepción de datos, dejando que la sección de la lógica programable transfiera grandes cantidades de datos con baja latencia.

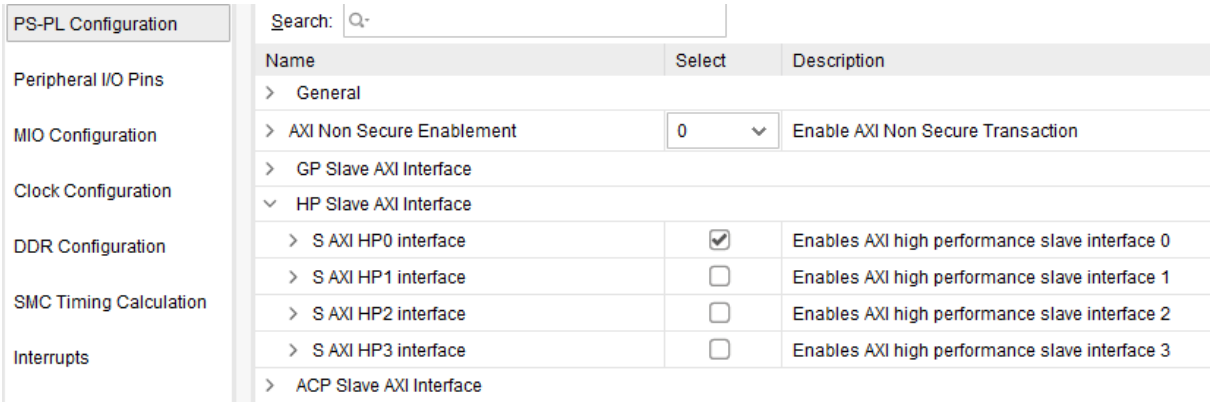

Figura 48. Configuración PS-PL

La Figura 49 muestra la pestaña correspondiente a los periféricos de entrada y salida (*I/O Peripherals*). Se activan solo los controladores indispensables para el diseño de la arquitectura HW de la plataforma inicial y el resto de los periféricos permanecen deshabilitados. En conjunto con la sección de configuración MIO de los *I/O Peripherals* (Figura 50), se habilitan los siguientes puertos:

- UART 0: Para la consola de depuración a 115 200 baudios en modo de configuración MIO 14 y 15 (UART_TX y RX).
- I2C 0 e I2C 1: Para el bus SCCB de configuración de las dos cámaras OV5640 del array via EMIO, enrutando SDA y SCL al conector RPi.
- GPIO: Para las señales RSTN y PWDN que necesitan los sensores OV5640 en el proceso de *power up*, así como el LED 0 de control/verificación de estado. Se enrutan vía EMIO, quedando disponibles en la matriz de pines de la sección PL.
- SD0: Para el almacenamiento de las secuencias RAW de los *buffers* de cada cámara en el proceso de la calibración *offline*. Se habilitan los pines 40-45 referentes a las señales CMD, DAT[3:0] y CLK.

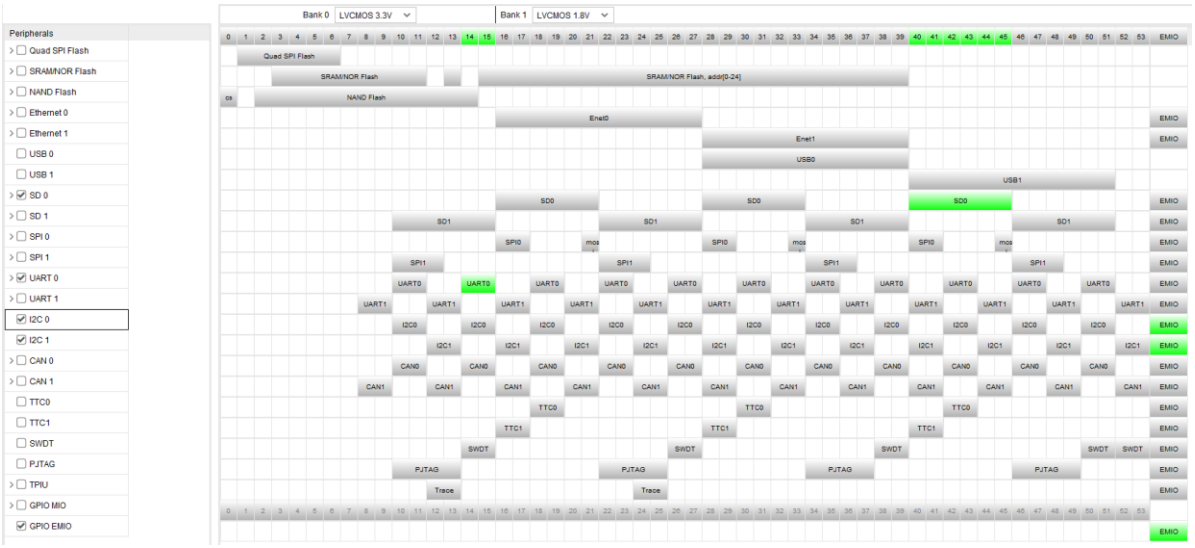

Figura 49. Pines periféricos de E/S

SD 0	MIO 40 .. 45					
CD						
WP						
Power						
SD 0	MIO 40	clk	LVCMOS 1.8V	slow	enable	inout
SD 0	MIO 41	cmd	LVCMOS 1.8V	slow	enable	inout
SD 0	MIO 42	data[0]	LVCMOS 1.8V	slow	enable	inout
SD 0	MIO 43	data[1]	LVCMOS 1.8V	slow	enable	inout
SD 0	MIO 44	data[2]	LVCMOS 1.8V	slow	enable	inout
SD 0	MIO 45	data[3]	LVCMOS 1.8V	slow	enable	inout
SD 1						
UART 0	MIO 14 .. 15					
UART 1						
I2C 0	EMIO					
I2C 1	EMIO					
SPI 0						
SPI 1						
CAN 0						
CAN 1						
GPIO						
GPIO MIO						
EMIO GPIO (Width)	3					

Figura 50. Configuración MIO

La pestaña *Clock Configuration* define la generación de relojes y *resets* del sistema, como se puede comprobar en la Figura 51. El reloj *PL Fabric* (FCLK) se activa con *CLK1* a una frecuencia de 100 MHz, que sirve de referencia para los buses AXI y para el IP *Dynamic Clock Generator.* Por otro lado, el reloj *CLK0* asignado a 50 MHz, se utiliza en el IP *Video Timing Controller* (s_axi_aclk), al *axi_dyn_clk* (s00_axi_aclk), y actúa como fuente de reloj para el IP *Clocking Wizard* (ver 4.3.13. IP).

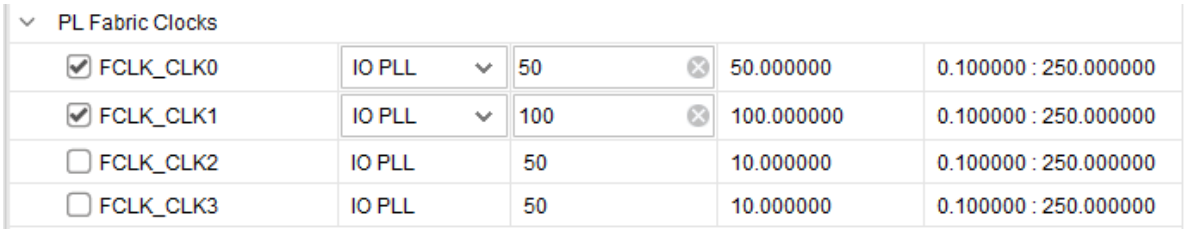

Figura 51. Configuraciones de reloj

4.3.13. IP *Clocking Wizard*

El array de cámaras PZ5640-D del fabricante Pú zhì no contiene ningún oscilador integrado que permita generar la señal de reloj precisa para el funcionamiento de las cámaras OV5640. En consecuencia, en este caso se ha tomado la señal de 50MHz generada por el *Fabric Clock 0* de la sección PS como referencia y se ha empleado el IP *Clocking Wizard* (Figura 52) para generar un reloj de 24MHz.

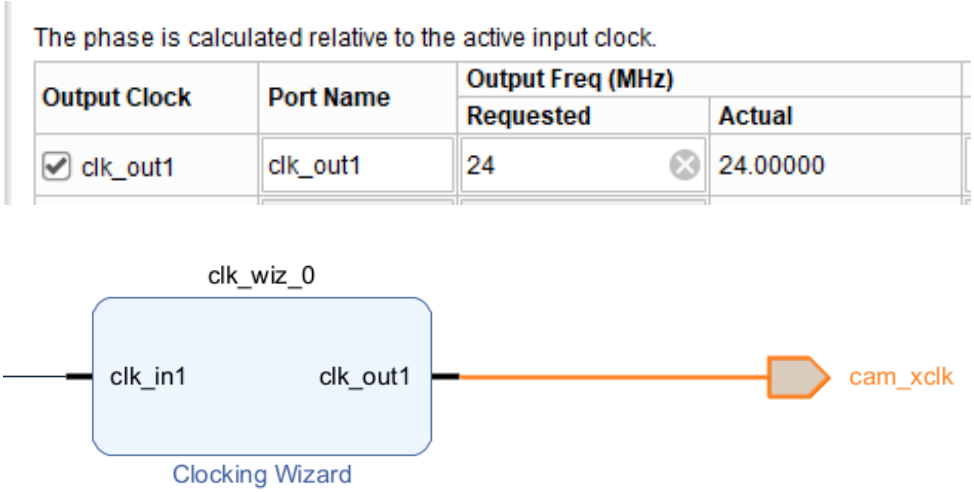

Figura 52. IP Clocking Wizard

La Figura 53 reproduce por consecuencia la plataforma de prueba para la visualización de las cámaras OV5640 desarrollado como paso previo a previa a la implementación del IP *stereoLBM*. Posteriormente se sustituirá el IP *Video Mixer* por el IP HLS *Stereolbm* y se eliminará el IP *Video Test Pattern Generator* para la correcta implementación del sistema completo.

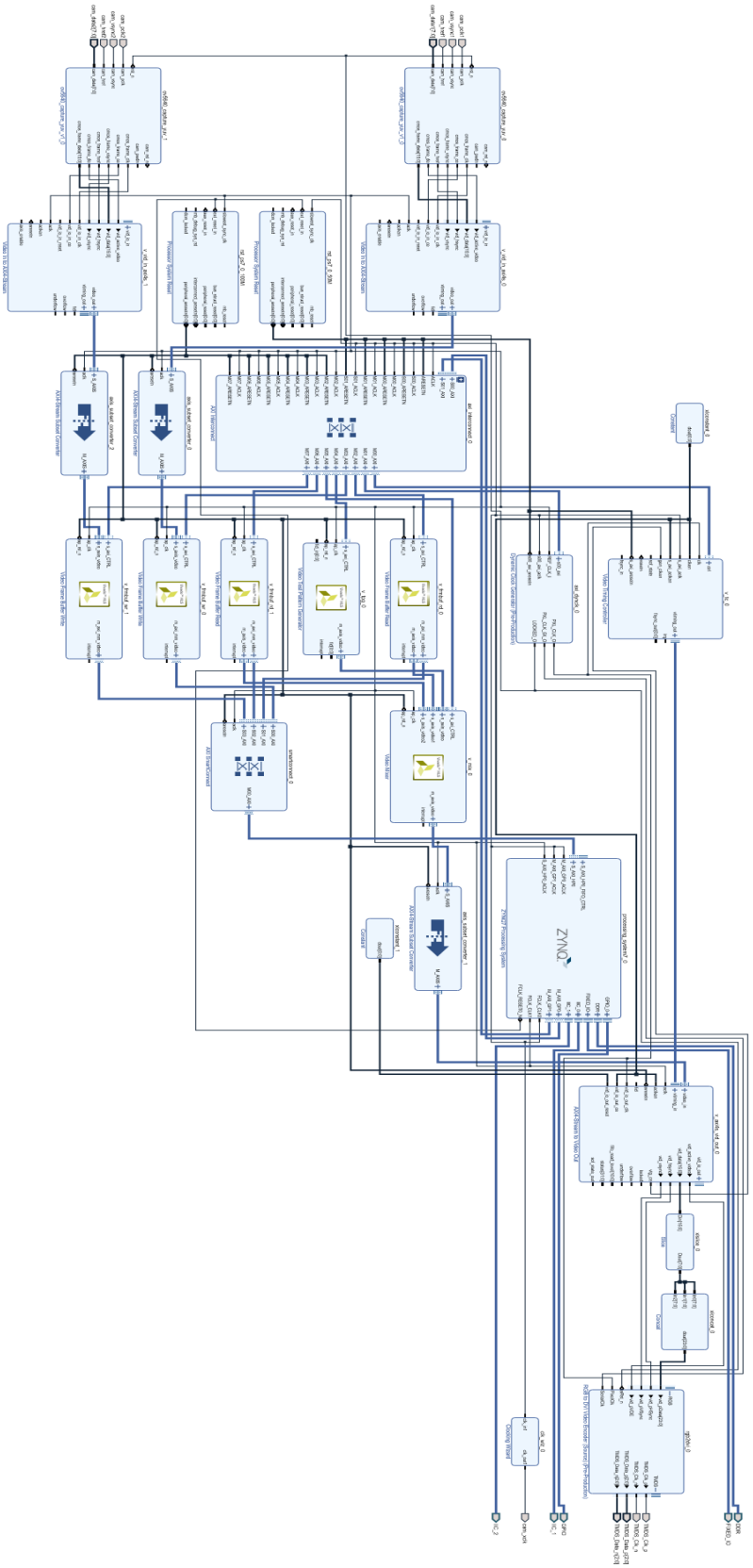

Figura 53. Integración final del hardware del array de cámaras YUV422

4.4. ARCHIVO DE RESTRICCIONES

El archivo TOP_pynqz2.xdc recoge la asignación de pines para todas las señales necesarias de la interfaz de entrada/salida (E/S) de la arquitectura *hardware* de la plataforma inicial de prueba, con las que se garantiza que la plataforma desarrollada funcione con la descripción de *hardware* resultante en la placa de prototipado PYNQ-Z2. La estructura lógica que se ha seguido para poder distinguirlas se recoge en la siguiente tabla.

Tabla 3. Resumen de restricciones de la plataforma

Función	Líneas	Correspondencia de pines
LED0	GPIO_tri_io[2]	R14
SEÑALES DE CONTROL RSTN Y PWDN	GPIO_tri_io[0](RSTN) GPIO_tri_io[1](PWDN)	U18 V7
Cámara 1	cam_data1[7:0] cam_pclk1 cam_vsync1 cam_href1 IIC_1_scl_io IIC_1_sda_io	V10, F20, C20, U8, V8, W6, Y7, U7 (D[7:0] respectivamente) Y6 (PCLK) V6 (VSYNC) Y18 (HREF/HSYNC) W19 (SCL1) W18 (SDA1)
Cámara 2	cam_data2[7:0] cam_pclk2 cam_vsync2 cam_href2 IIC_2_scl_io IIC_2_sda_io	Y9, W8, A20, W9, U19, Y19, B20, Y8 (D[7:0] respectivamente) F19 (PCLK) W10 (VSYNC) B19 (HREF/HSYNC) Y16 (SCL2) Y17 (SDA2)
Reloj externo	cam_xclk	T14
Salida HDMI – reloj TMDS	TMDS_Clk_n TMDS_Clk_p	L17 L16
Salida HDMI – datos TMDS	TMDS_Data_n[2:0] TMDS_Data_p[2:0]	K18, J19, H18 K17, K19, J18

La topología completa, con la nomenclatura Vivado del conector RPi de la placa de prototipado PYNQ-Z2, se ilustra en la Figura 54. Toda la lógica opera en LVCMOS33, a excepción de los cinco pares de canales TMDS que funcionan con el *IO Standard* TMDS_33, siguiendo la compatibilidad de la capa física de HDMI.

G	W9	Y8	W8	Y7	Y6	Y16	G	W10	V10	V8	V	U8	V7	U7	G	V6	W19	W18	V
39	37	35	33	31	29	27	25	23	21	19	17	15	13	11	9	7	5	3	1
40	38	36	34	32	30	28	26	24	22	20	18	16	14	12	10	8	6	4	2
Y9	A20	B19	G	B20	G	Y17	F20	F19	U19	G	U18	W6	G	C20	Y19	Y18	G	V	V

Figura 54. Disposición de pines del encabezado de Raspberry Pi y asignaciones de pines de Zynq PL

Los relojes de píxel *cam_pclk1* y *cam_pclk2* están asignados a pines de la PL con capacidad MRCC (*Multipurpose Routed Clock Capable*), que permiten rutas de reloj dedicadas y de bajo *skew*. Sin embargo, Vivado sugiere ubicar un *buffer* global de reloj (BUFG) en un semicuadrante distinto al de esas señales, lo que provoca advertencias de enrutamiento al exigir el uso de rutas de reloj dedicadas que no pueden cumplirse físicamente.

Para garantizar que ambos relojes lleguen correctamente al dominio *AXI-Stream*, se desactivan esas restricciones con las siguientes directivas en el XDC:

- `set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets cam_pclk1_IBUF]`
- `set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets cam_pclk2_IBUF]`

Con ello, Vivado deja de forzar el uso de las rutas de reloj dedicadas para esos *nets*, evitando fallos de enrutado y asegurando la integridad y sincronización de las señales de píxel. También se activan las resistencias de *pull-up* para los buses I^2C, que evita componentes discretos adicionales en el conector RPi de la placa de prototipado.

Para asignar correctamente los pines del array de cámaras PZ5640-D sobre la placa de prototipado PYNQ-Z2, en la siguiente tabla se recoge la correspondencia entre pines del conector JM1 y los pines del conector Raspberry Pi.

Tabla 4. Conexion pin JM1 del array de cámaras con RPi-pynq.

Función (OV5640)	Pin JM1	Pin RPi-PYNQ	Nombre del esquemático RPi
Alimentación	2	1	3V3
Masa	3, 4, 33, 34, 35, 36	6, 9, 25, 30, 34, 39	GND1-6
BUS I²C cámara 1			
SCL1	21 (CAM1_IIC_SCK)	5	SCL1
SDA1	22 (CAM1_IIC_SDA)	3	SDA1
BUS I²C cámara 2			
SCL2	5 (CAM2_IIC_SCK)	27	SCL2
SDA2	6 (CAM2_IIC_SDA)	28	SDA2
Señales de control comunes			
RESET (n)	40 (CAM_nRST)	31	RSTN
PWDN	— (Conectado a tierra)	13	PWD
XCLK 24 MHz	20 (CAM_24 MHz)	-	-
Cámara 1			
PCLK	37	10	PCK1
VSYNC	38	8	VS1
HSYNC/HREF	39	7	HR1
D0 a D7	29, 31, 32, 23, 30, 28, 27, 26	11, 18, 16, 19, 15, 12, 22, 21	D10 a D17
Cámara 2			
PCLK	17	24	PCK2
VSYNC	18	23	VS2
HSYNC/HREF	9	36	HR2
D0 a D7	13, 15, 16, 7, 14, 12, 11, 10	35, 32, 29, 26, 37, 38, 33, 40	D20 a D27

En la Figura 55 se aprecia el array de sensores OV5640, conectados a la placa de prototipado PYNQ-Z2 a través del conector JM1. Los ocho pares de líneas de datos D0-D7 de cada cámara, se conectan a la placa dejando a un lado los correspondientes a D8 y D9.

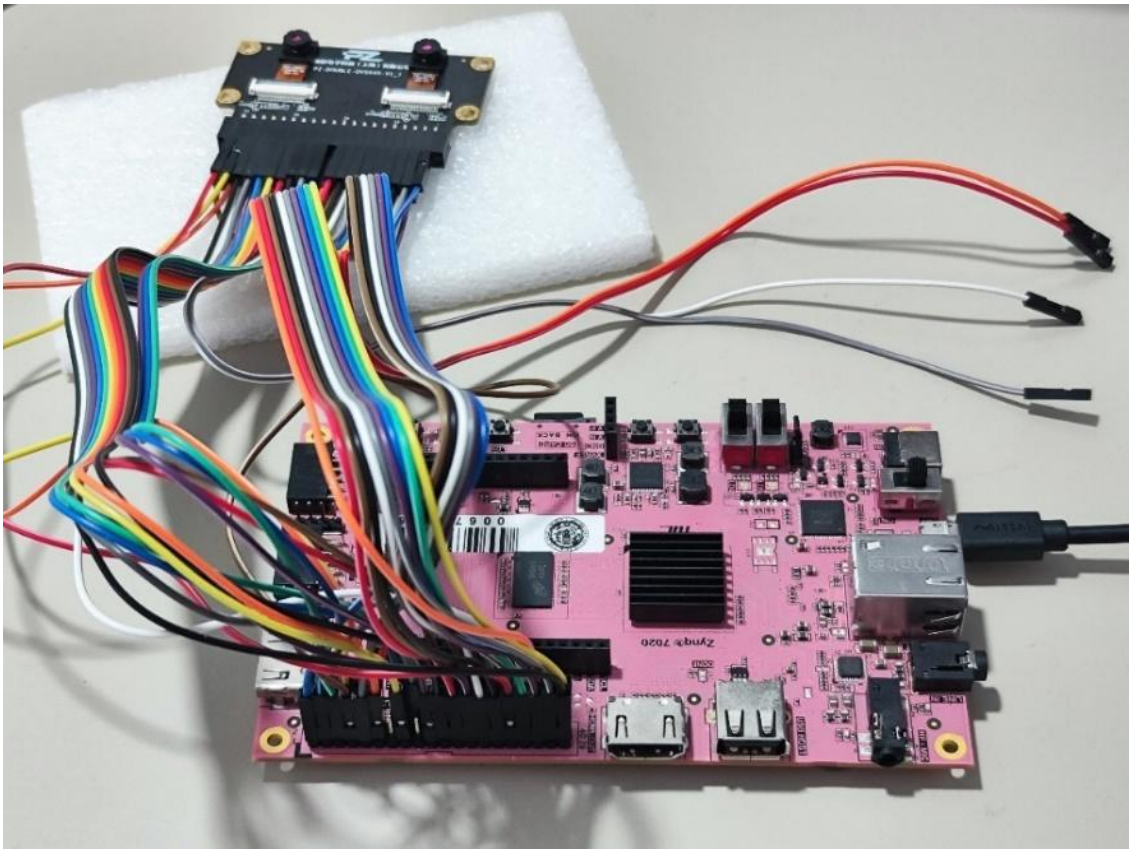

Figura 55. Motaje del array de cámaras PZ5640-D sobre la placa PYNQ-Z2

4.5. DISEÑO DE PCB

Para evitar el cableado manual cada vez que se quiera verificar el uso de un *array* de cámaras, y mejorar la estructura y robustez del diseño, se propuso el desarrollo de una PCB que enrute el conector RPi de la placa de prototipado PYNQ-Z2 con el conector JM1 del *array* de cámaras.

Como la plataforma se ha validado con varios *arrays* de cámaras de fabricantes diferentes que integran el modelo OV5640 en su diseño, se ha optado por desarrollar una solución que soporte doble conexión, para evitar tener que fabricar una PCB por cada modelo. Así, tanto el modelo PZ5640-D como el ATK-OV5640-DUAL, pueden ser evaluados a lo largo del proceso de creación de la plataforma.

Se ha elegido la herramienta KiCad para realizar el diseño de la PCB, ya que es un paquete de *software* libre ampliamente usado en el diseño de esquemáticos y placas de circuitos impresos.

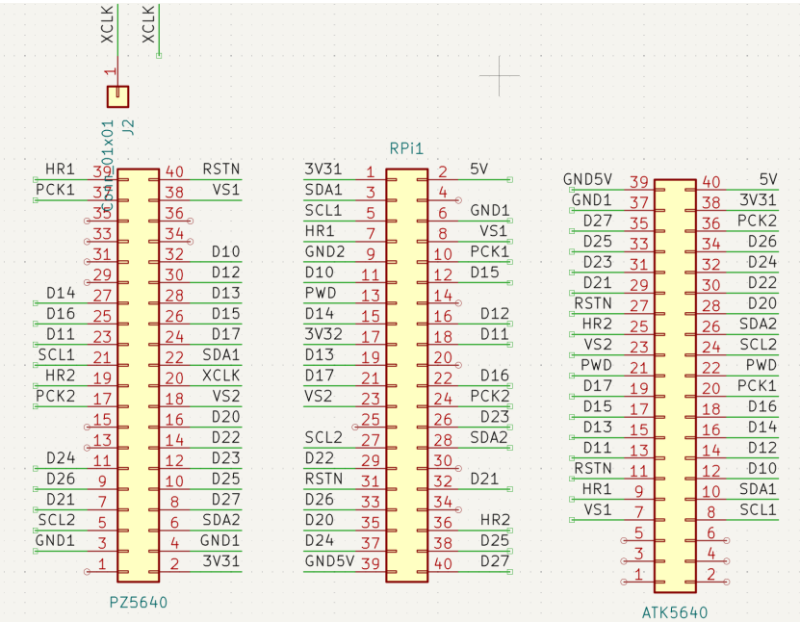

Figura 56. Esquemático del diseño para la interconexión del array de cámaras OV5640

La Figura 56 muestra el esquemático de la PCB con la distribución de pines para cada conector, en la función del módulo del array de cámaras. Cabe destacar que en el esquemático del conector PZ5640-D, el pin 20 se enruta a un jumper externo de la PCB debido a que el array de cámaras de este fabricante no tiene soldado el oscilador para generar la señal de reloj de 24MHz, pese a que lo indica en el esquemático. Esta señal se genera internamente desde la sección PS del dispositivo ZYNQ integrado en la placa de prototipado PYNQ-Z2. Se toma como referencia una señal de 50MHz generada por el *Fabric Clock 0* y empleando un IP *Clocking Wizard* para derivar un reloj de 24MHz y así poder asociar esta señal a un pin Arduino disponible.

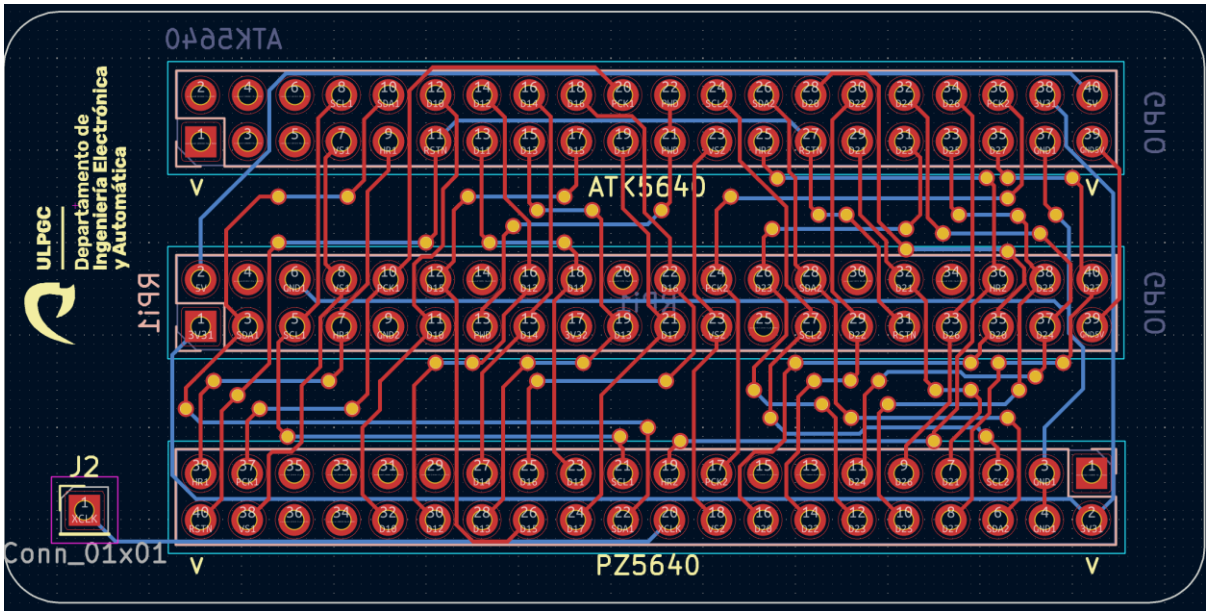

Figura 57. Vista de ruta de pistas – Capa de cobre superior e inferior

La Figura 57 muestra la configuración del trazado de pistas realizado a 2 capas, superponiendo la capa de cobre superior (TOP) de color rojo y la capa inferior (BOTTOM) representada en azul. Se ha seleccionado una PCB de doble capa fabricada en FR-4 de 1,51 mm, con un grosor de cobre de 35 µm y una máscara de soldadura verde convencional. El grosor mínimo de pista es de 0,20 mm.

Para ajustarse al espacio disponible en la placa de prototipado PYNQ-Z2 y mantener un diseño compacto, la placa final mide 6,5 cm de largo, 3,2 cm de ancho, y la distancia entre los dos conectores en paralelo es de 1,5 cm.

La Figura 58 muestra el render 3D de la PCB generada en KiCad, con los tres conectores soldados en paralelo y la Figura 59 muestra como referencia el módulo dual PZ5640-D montado sobre la placa de prototipado PYNQ-Z2 a través de la PCB implementada.

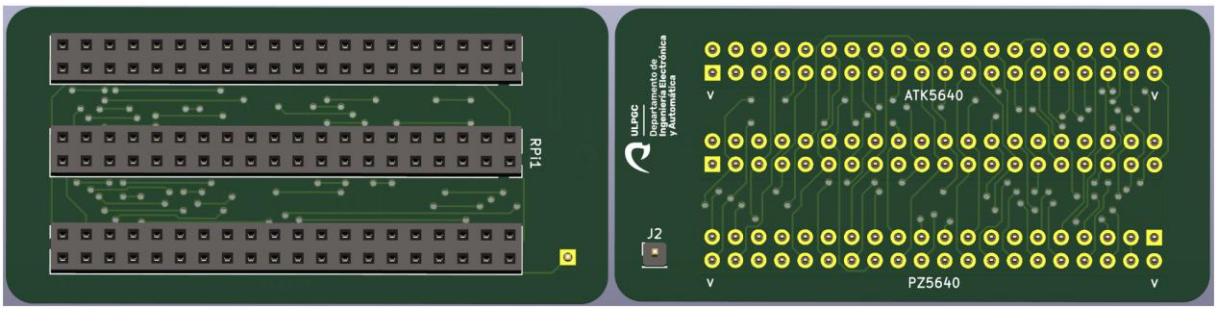

Figura 58. Render 3D de la PCB

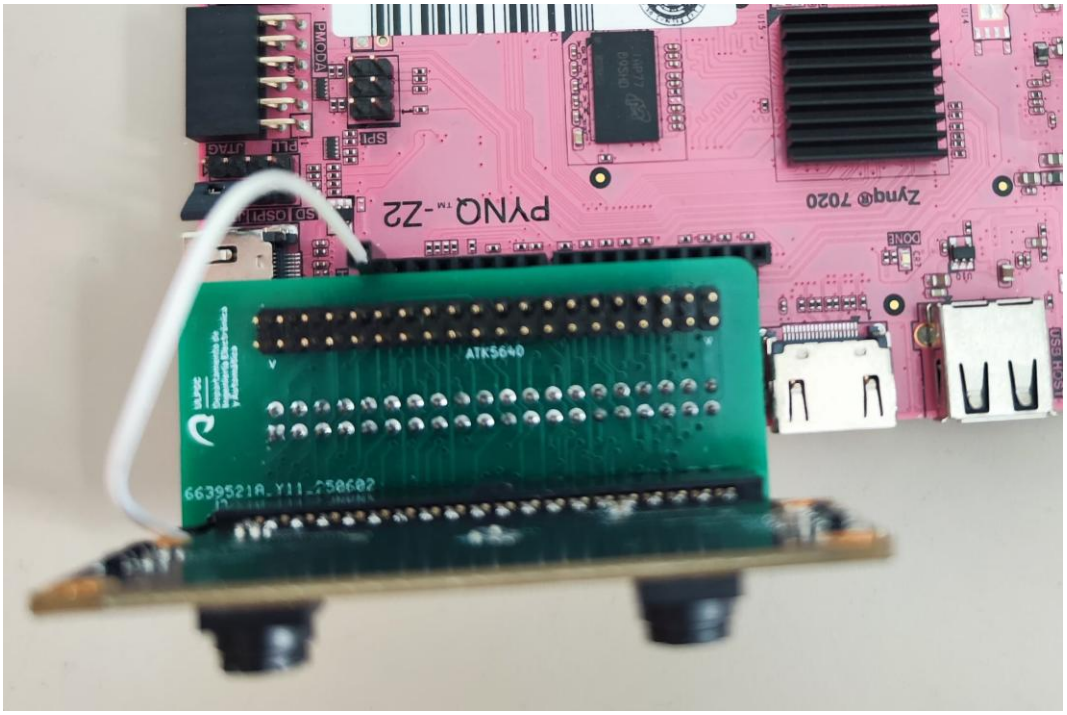

Figura 59. Módulo dual sobre la PYNQ-Z2

4.6. CADENA COMPLETA DE CAPTURA EN LA PLATAFORMA DE PRUEBA CON FORMATO DE VÍDEO YUV422

La cadena de captura representa la parte funcional de la plataforma inicial, puesto que une físicamente las dos cámaras OV5640 con la lógica programable y, a su vez, con la memoria DDR y la salida HDMI. En la Figura 60 se presenta la arquitectura de alto nivel que se ha implementado en Vivado: desde la interfaz DVP de 8 bits de cada cámara, pasando por los píxeles enviados al IP de captura, que los convierte en un flujo de datos *AXI4-Stream* y posteriormente se almacenan en DDR a través de los bloques IP *Video Frame Buffer*

Write/Read. Por último, los datos se envían al *Video Mixer* y, finalmente a la salida de este subsistema HDMI, para su visualización en tiempo real.

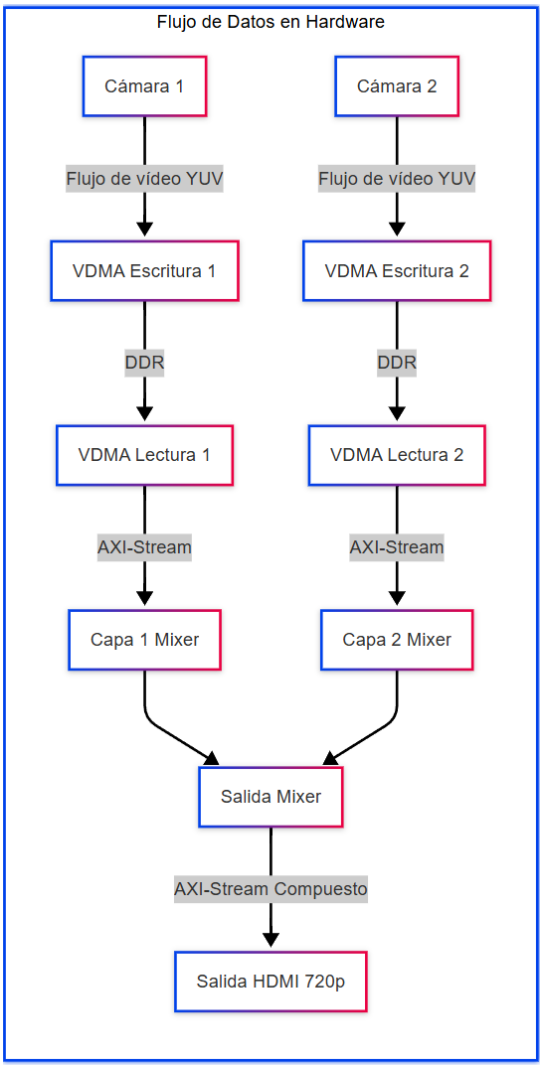

Figura 60. Arquitectura de alto nivel del sistema.

4.6.1. Resultados tras la implementación

Tal y como se observa en la Figura 61, el informe de potencia que se obtiene tras la implementación de la arquitectura HW de la plataforma inicial refleja un consumo total *on-chip* de 2.418 W, de los cuales un 93% corresponde a consumo dinámico. El núcleo PS7 concentra algo más de la mitad de éste, seguido por los módulos de lógica programable y el resto de los bloques que apenas superan el 7% combinado. La temperatura de unión se mantiene en torno a

53 °C bajo condiciones de operación típicas, lo que corrobora el gran margen térmico de la del dispositivo ZYNQ-7000 incluso con ambos sensores activos a 60 fps.

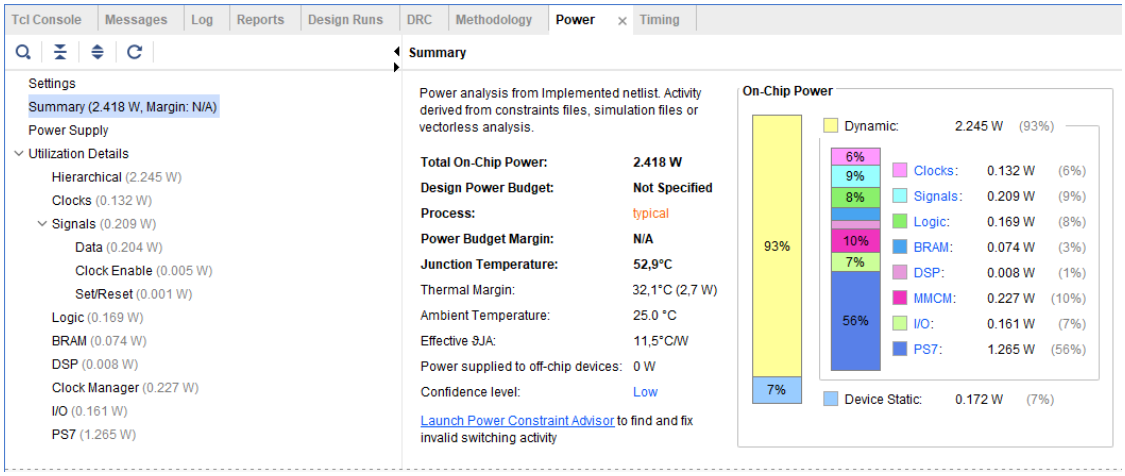

Figura 61. Informe resumido de consumo de potencia tras la implementación.

En la Figura 62 puede observarse que la utilización de recursos una vez implementado el diseño es de aproximadamente un: 65 % de LUT, un 39 % de FF, un 35 % de BRAM, un uso de DSP del 5 % y un uso de MMCM del 50 %. Esta distribución resultante da un margen bastante justo para poder integrar el acelerador SLBM. Se debe mencionar que la ocupación de I/O es del 30 %, valor lógico teniendo en cuenta las líneas dedicadas a las cámaras, al bus HDMI y a la interfaz de depuración.

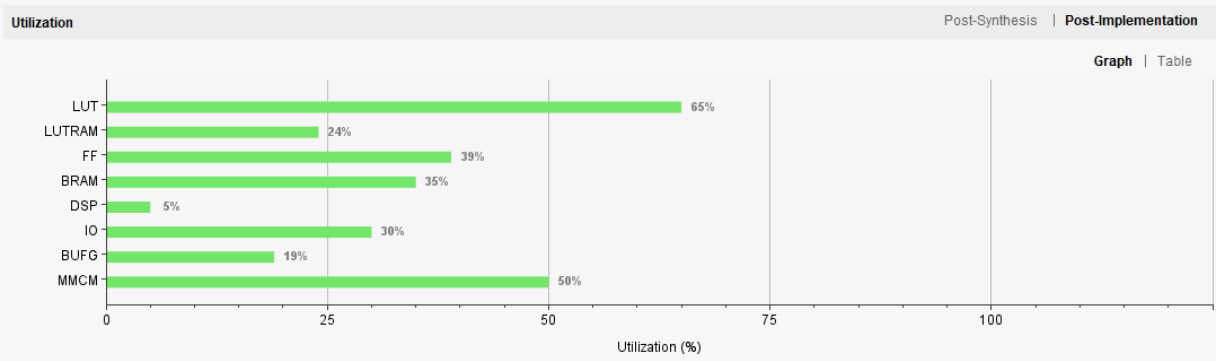

Figura 62. Utilización de recursos post implementación

La Figura 63 muestra el diseño implementado, el cual sirve como banco de referencia de rendimiento y consumo. Así, se puede medir posteriormente el impacto del IP de disparidad SLBM (Capítulo 5) a partir de su implementación.

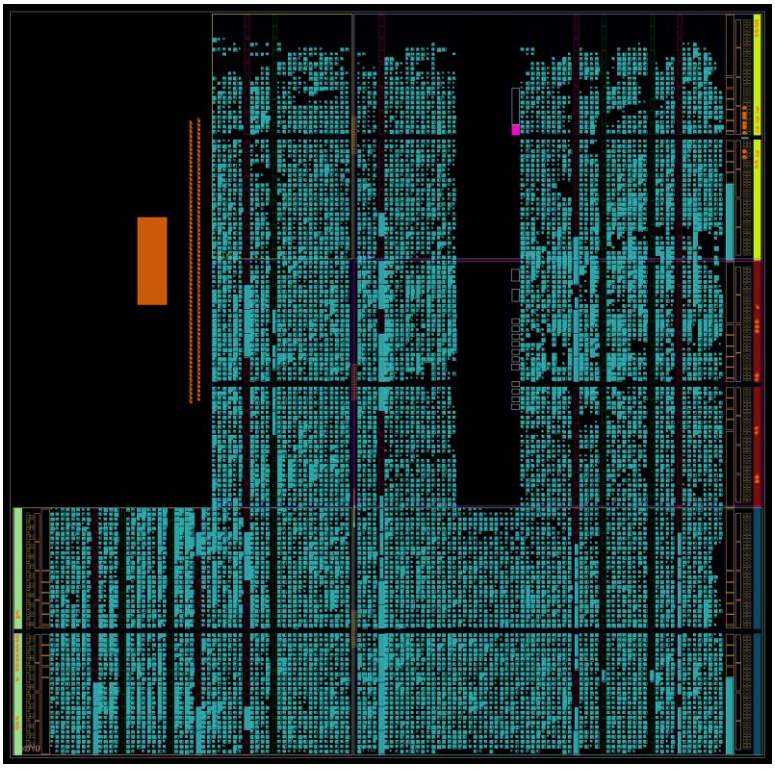

Figura 63. Diseño implementado pre IP SLBM

La Figura 64 reúne las métricas obtenidas en cuanto a temporización después de la implementación para una señal de reloj de 100 MHz ($T_{100MHZ} = 1/100MHz = 10\,ns$). El informe que genera Vivado no presenta violaciones ni de *setup* ni de *hold*, indicando que la especificación se cumple con un buen margen. Concretamente, atendiendo al *worst negative slack* (*WNS*) de 1,419 ns se puede determinar la frecuencia máxima teórica de operación del sistema.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1,419 ns	Worst Hold Slack (WHS):	0,025 ns	Worst Pulse Width Slack (WPWS):	0,333 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	79885	Total Number of Endpoints:	79228	Total Number of Endpoints:	32129

All user specified timing constraints are met.

Figura 64. Resumen del timing del diseño obtenido para un reloj de 100MHz tras la implementación

$$T_{\max} = T_{100MHZ} - wns = 10 - 1{,}419 = 8{,}581ns \quad (2)$$

Por tanto,

$$f_{\max} = \frac{1}{8{,}581ns} \approx 116{,}50\ MHz \quad (3)$$

donde $T_{\max}$ representa el periodo para la máxima frecuencia teórica, $f_{\max}$ la máxima frecuencia teórica y wns es el *worst negative slack*.

Partiendo de la frecuencia teórica máxima estimada y de la frecuencia finalmente configurada de 100MHz, el margen se calcula de la siguiente manera:

De (2) se despeja

$$T_{\max} = T_{100MHZ} - wns \Rightarrow wns = T_{100MHZ} - T_{\max} = 1{,}419ns$$

y se sabe que

$$T_{\max} = 8{,}581ns$$

Por lo que el cálculo del porcentaje de $wns_{\%}$ da como resultado:

$$wns_{\%} = \frac{1{,}419 \cdot 100}{8{,}581} \approx 16{,}5\% \quad (4)$$

Es decir, el diseño tiene una holgura en su ruta crítica del 16,5% operando a una frecuencia de 100 MHz. Por último, cabe destacar que el *Total Negative Slack (TNS)* y el *number of failing Endpoints* se mantienen en 0,000 ns y 0 rutas (Figura 64), respectivamente, lo que ratifica la ausencia de rutas fallidas. De esta forma, la infraestructura *hardware* puede operar sostenidamente a 100 MHz sin comprometer la fiabilidad del sistema ni tampoco la integridad temporal del flujo de datos registrados.

Con la infraestructura ya consolidada, el siguiente paso consiste en desarrollar la aplicación *software* que se ejecutará en la sección PS del dispositivo ZYNQ con el fin de mostrar la captura simultánea de las imágenes de los sensores OV5640 de los *arrays* de cámaras, y posteriormente integrar el acelerador SLBM sobre la lógica programable, objetivo que se aborda en el Capítulo 5.

4.7. APLICACIÓN SOFTWARE

4.7.1. Creación de una aplicación a partir de un archivo .xsa

Una vez finalizado el diseño de la arquitectura *hardware* de la plataforma inicial en el entorno Vivado 2023.1, se exporta la plataforma *hardware* mediante un archivo .xsa (*Xilinx Support Archive*). Este archivo es un contenedor que engloba toda la información del diseño *hardware* y que es necesario para el entorno de *software* en el que se incluye entre otros elementos:

- El *bitstream* de la FPGA.
- La configuración de inicialización del procesador PS Zynq.
- La descripción de los periféricos AXI con su respectiva asignación de direcciones de memoria.

Al exportar la arquitectura *hardware* en esta fase de *handoff*, se comprueba que se incluye el *bitstream* que se genera [35].

En la fase de desarrollo de *software bare-metal* se importa el archivo XSA en la herramienta AMD Vitis 2023.1. Al crear un nuevo proyecto de plataforma en Vitis basado en el fichero XSA exportado, la herramienta genera automáticamente un *Board Support Package* (BSP) para el diseño *hardware* desarrollado. El BSP, es un paquete de soporte que contiene los drivers de más bajo nivel, las bibliotecas y las definiciones (como la de *xparameters.h*) que la aplicación C/C++ necesita para funcionar con los periféricos que se hayan definido en el diseño *hardware*. Posteriormente, se gestiona el proyecto de aplicación C *bare-metal* asociado a la plataforma *hardware* que se está generando. En esta aplicación se implementa el código fuente (*main.c*) para inicializar los dispositivos y llevar a cabo la lógica de la aplicación. El termino *bare-metal* indica que la aplicación se ejecuta sobre el procesador sin necesidad de ningún sistema operativo, utilizando las funciones que ofrece el BSP (*standalone*) para interactuar con los periféricos. Así, la aplicación *software* puede configurar registros de control (vía interfaces *AXI-Lite*) y mover datos desde la memoria con los *framebuffers,* entre otras funciones.

El funcionamiento del sistema se muestra en el diagrama de flujo de la Figura 65. Éste se divide fundamentalmente en dos fases.

La "Fase 1: Inicialización y Configuración" únicamente se ejecuta al inicio y su objetivo es preparar el *hardware* ya que, en este punto, se inicializan los módulos periféricos y los sensores de las cámaras. A continuación, se ejecuta un paso de verificación crítica: se espera a recibir de ambas cámaras una respuesta con su ID I2C correcto. Si esta condición no se verifica, el sistema hace saltar un error y se aborta la actividad para poder facilitar la depuración. Si se recibe la respuesta esperada, se configura secuencialmente todo el *pipeline* de vídeo: se programan los registros de las cámaras 1 y 2, se enruta la señal HDMI a la tarea de grabación y se inicia el IP TPG. Por último, se inserta el esquema de doble *buffer* (*ping-pong*) y el IP *Video Mixer*.

Una vez el sistema está totalmente configurado, la aplicación entra en la "Fase 2: Bucle de Operación Principal" que se ejecuta indefinidamente y es la responsable de gestionar el flujo de vídeo en tiempo real. El *software* comprueba en cada iteración y para cada cámara, si el *buffer* de escritura ha completado la captura de un fotograma completo. En caso positivo, se realiza el intercambio "*ping-pong*" donde el *buffer* potencialmente lleno pasa a ser el que se encuentra en la pantalla, y el anterior queda libre para almacenar el fotograma siguiente. Este ciclo de comprobación e intercambio continuo de *buffers* garantiza que la salida de vídeo sea correcta, fluida y sin artefactos visuales como el *tearing*, cumpliendo así con el objetivo de la aplicación *software*.

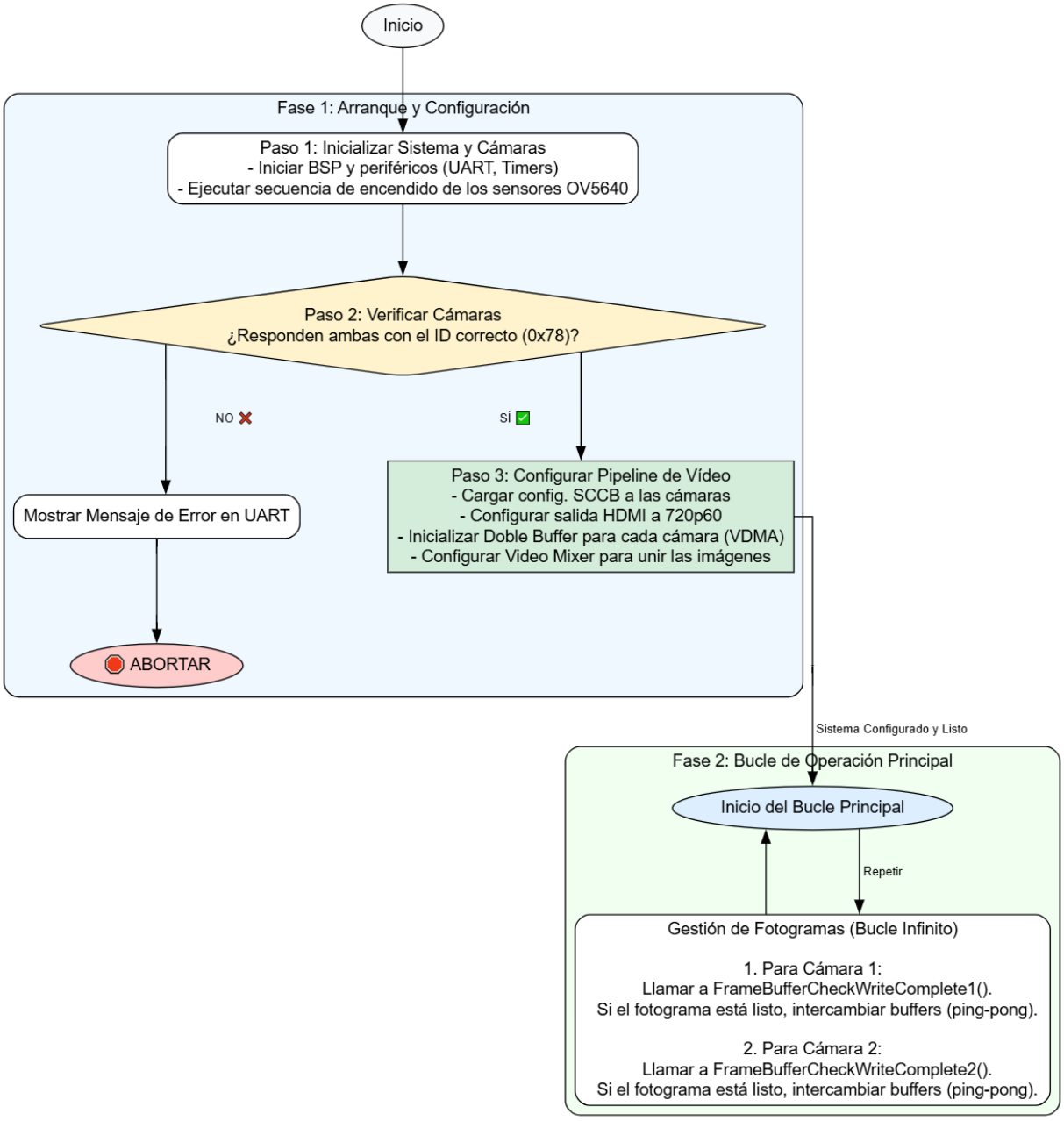

Figura 65. Diagrama de flujo genérico del sistema

A fin de que el proyecto *software* desarrollado sea fácil de seguir y reproducir, el código se estructura en una jerarquía de carpetas que separa bien cada uno de los bloques funcionales, lo que ayuda a tener una estructura modular que favorece la reutilización de elementos y la depuración de la aplicación. Dicha estructura es la siguiente:

- Fichero *ov5640_sccb*: *ov5640_sccb.c* y *ov5640_sccb.h*. Controlador SCCB cuyo cometido es la inicialización y configuración de los sensores OV5640 mediante el bus I²C.
- Fichero *display_ctrl_hdmi*: *display_ctrl.c*, *display_ctrl.h*. Generación del *timing* HDMI y selección de modos de vídeo; *lcd_modes.h* contiene las tablas de resoluciones y sincronismos.
- Fichero *dynclk*: dynclk.c, *dynclk.h*. API para programar el *IP Dynamic Clock Generator*, que permite cambiar la frecuencia de píxel sin tener que re-sintetizar la lógica de la sección PL.
- Fichero mixer: *mixer.c*, *mixer.h*. Abstracción y gestión del IP *Video Mixer*. Su función es tomar dos o más flujos de vídeo y combinarlos por capas mediante *alpha-blending*.
- Fichero *tpg*: *tpg_ctrl.c, tpg_ctrl.h*. Control del IP *Video Test Pattern Generator*, utilizado para depuración del canal de vídeo cuando no hay cámaras conectadas.
- Fichero *vframebuffers*: *vframebuffers.c, vframebuffers.h*. Reserva de DDR, configuración de los IP *Video Frame Buffer*, gestión *ping-pong* de fotogramas.
- Fichero *main.c*: punto de entrada de la aplicación. Se encarga de inicializar el BSP y los módulos de vídeo antes de entrar en el bucle infinito principal de la aplicación.
- Ficheros *platform: platform.c, platform.h, platform_config.h*. Utilidades genéricas (UART, temporizadores, gestores de interrupciones).
- Fichero *lscript.ld*: *linker script* elaborado a partir del mapa de memoria del diseño.
- Fichero *Xilinx.spec*: configuración del *toolchain* que es utilizada por *Vitis* para la construcción cruzada.
- Ficheros de salida: directorios *Binaries*, *Includes* y *Debug* que contienen artefactos de compilación, objetos intermedios y dependencias.

4.7.2. Gestión de las cámaras

La gestión de control software de cada una de las cámaras OV5640 queda encapsulada a su *driver ov5640_sccb.c/h*, el cual se encarga de:

- Inicializar los controladores I²C PS-IIC por cada cámara y las líneas GPIO de RESET y PWDN.
- Implementar primitivas de lectura/escritura sobre el bus SCCB ⇔ I²C.
- Emplear las secuencias de *power-up*, puesta en modo *stand-by*, carga de registros y salida a flujo activo.

La gestión de las cámaras sigue la secuencia de funcionamiento que se describe en los siguientes apartados.

4.7.2.1. Inicialización.

La función *OV5640_Init()* ejecuta la secuencia temporal que el fabricante determina como recomendable (mencionado en el apartado

3.2.1. Secuencia de inicialización del sensor) para asegurar que el sensor OV5640 se inicia siempre en un estado conocido. De forma esquemática, realiza los siguientes pasos:

1. Forzar *reset* y mantener sensor apagado:
   - Coloca la señal RST_N a “0” (*reset* activo).
   - Lleva la señal PWDN a “1” (modo *power-down*), desconectando internamente el sensor.
2. Aplicar tensiones de alimentación:
   - Enciende primero la tensión de alimentación digital (DOVDD) y, a continuación, la tensión de alimentación analógica (AVDD), según las recomendaciones del fabricante.
3. Salir de *power-down*:
   - Establece la señal PWDN al nivel “0” lógico, permitiendo que el sensor abandone el modo de bajo consumo.
4. Liberar *reset*:

- Establece la señal RST_N al nivel “1” lógico, iniciando el posicionamiento interno de todos los bloques lógicos del sensor.

5. Esperar disponibilidad del bus SCCB:

   - Antes de empezar a escribir los registros de configuración, la rutina comprueba que el controlador I²C interno (bus SCCB) esté libre y listo para la transferencia de datos.

4.7.2.2. Comprobación de ID I²C .

En la función *main()* del fichero *main.c* se evalúa el valor del registro 0x3100 correspondiente al ID del sensor OV5640 (Código 3). El sistema aborta con un mensaje UART si cualquiera de las cámaras devuelve un identificador diferente del valor 0x78 especificado por el fabricante, facilitando la detección de fallos de cableado o bus antes de poder continuar con el flujo de captura y procesamiento de vídeo.

Código 3. Comprobación del ID del sensor

```
u8 ov5640_id1 = SCCB1_RecvByteFReg(0x3100);
if (ov5640_id1 != 0x78) {
    xil_printf("!!! ov5640_1 no encontrado, ID = (0x%X)\n\r", ov5640_id1);
    return -1;
} else {
    xil_printf("ov5640_1 encontrado (%X)\n", ov5640_id1);
}
```

4.7.2.3. Carga de la configuración SCCB.

La función *OV5640_Config()* recorre dos tablas de pares registro-dato: *ov5640_setting_VGA_640_480_YUV422[]* y *ov5640_setting_HD_1280_720_YUV422[]*. Estas tablas fijan de forma secuencial más de 180 pares registro-dato para la configuración de cada uno de los sensores OV5640. Entre los registros clave que están se configuran, se puede ver en la Figura 66 el mapeo de registros implicados en la selección del formato. Se resalta el registro asociado al formato YUV422 con la secuencia YUYV (`{0x4300, 0x30}`).

table 6-5 FORMAT control registers (sheet 1 of 5)

address	register name	default value	R/W	description
0x4300	FORMAT CONTROL 00	0xF8	RW	Format Control 00 Bit[7:4]: Output format of formatter module 0x0: RAW Bit[3:0]: Output sequence 0x0: BGBG... / GRGR... 0x1: GBGB... / RGRG... 0x2: GRGR... / BGBG... 0x3: RGRG... / GBGB... 0x4~0xF: Not allowed 0x1: Y8 Bit[3:0]: Does not matter 0x2: YUV444/RGB888 (not available for full resolution) Bit[3:0]: Output sequence 0x0: YUVYUV..., or GBRGBR... 0x1: YVUYVU..., or GRBGRB... 0x2: UYVUYV..., or BGRBGR... 0x3: VYUVYU..., or RGBRGB... 0x4: UVYUVY..., or BRGBRG... 0x5: VUYVUY..., or RBGRBG... 0x6~0xE: Not allowed 0xF: UYVUYV..., or BGRBGR... 0x3: YUV422 Bit[3:0]: Output sequence 0x0: YUYV...

Figura 66. Registros de control de formato

Por otro lado, para determinar la frecuencia interna de las señales de reloj del sensor OV5640, es necesario configurar los registros de control del PLL y los divisores para el reloj del sensor. En el modo normal, los bloques funcionales y los relojes internos se habilitan mediante la carga de los registros `{0x3004, 0xFF}` (habilita todos los relojes posibles) y `{0x3006, 0xC3}` (deshabilita el reloj JPEG); simultáneamente se carga el registro `{0x302E, 0x00}` en función de la configuración con la propia interfaz. Una vez ejecutadas estas inicializaciones, se selecciona el reloj del sistema que proviene del PLL, en el registro {0x3103, 0x03} (bit[1] = 1).

El ajuste de los registros del PLL se lleva a cabo mediante las máquinas de estado correspondientes, `{0x3035, 0x41}` y `{0x3036, 0x69}`, que modifican el multiplicador del PLL hasta conseguir la frecuencia deseada en el sensor. Se introduce el registro `{0x3108, 0x01}` que configura el multiplicador raíz de los PLL y de los divisores SCLK y PCLK (bits[5:4] y [1:0] respectivamente).

El registro `{0x3108, 0x01}` configura los divisores raíz de los PLL y de las divisiones SCLK y PCLK (bits [5:4], y, [1:0], respectivamente), configurados como transmisión por

defecto a las frecuencias especificadas en el *datasheet*, de esta forma quedan habilitados los registros para el funcionamiento de *pixel clock* (PCLK) emulando el modo manual del divisor PCLK del modo como se indica con el valor `{0x460B, 0x35}` y posteriormente el valor del registro `{0x460C, 0x22}`.

Seguidamente se especifica el factor de división que se utiliza mediante la carga del valor de registro `{0x3824, 0x02}` (modo VGA) o `{0x3824, 0x04}` (modo HD) según la tabla de valores de división que se contempla en la configuración para el cómputo de división. De esta forma el sensor OV5640 configurado mantendrá la configuración del reloj interno programando los modos VGA y HD para el *pixel clock* y su correspondiente interrupción.

Por otra parte, el tamaño de *frame* de salida se define mediante los registros de ancho (DVPHO) y alto (DVPVO) de la ventana de lectura. Para el modo VGA (640×480) se usan los valores (DVPHO = 0x0280 = 640) y (DVPVO = 0x01E0 = 480). Para el modo HD 720p (1280×720) los registros se fijan a (DVPHO = 0x0500 = 1280) y (DVPVO = 0x02D0 = 720). Estas entradas de registro definen la ventana de salida deseada para cada modo de vídeo.

Como referencia, en la Tabla se recogen los registros clave de configuración mencionados anteriormente:

Registros {Dirección, Valor}	Utilidad
{0x3004, 0xFF}	Habilita todos los relojes del sensor.
{0x3006, 0xC3}	Deshabilita relojes de bloques JPEG.
{0x3103, 0x03}	Selecciona el reloj del PLL como fuente de sistema.
{0x3035, 0x41}, {0x3036, 0x69}	Ajustan el multiplicador del PLL (configuración interna de clock).
{0x3108, 0x01}	Configura divisores raíz de PCLK/SCLK según la tabla de divisores del OV5640.
{0x460B, 0x35}, {0x460C, 0x22}	Activa el modo manual del divisor PCLK (bit[1]=1).
{0x3824, 0x02} (VGA) {0x3824, 0x04} (HD)	Factor de división final del PCLK en modo manual.
{0x3808, 0x02}, {0x3809, 0x80}, {0x380A, 0x01}, {0x380B, 0xE0}	Resolución VGA 640×480.
{0x3808, 0x05}, {0x3809, 0x00}, {0x380A, 0x02}, {0x380B, 0xD0}	Resolución HD 1280×720.
{0x4300, 0x30}	Selección del formato YUV422 (YUYV).
{0x501F, 0x00}	Multiplexor de formato en modo YUV422.

Tabla 5. Registros calve de configuración

4.7.2.4. Topología de buses.

Cada cámara del array se encuentra conectada en un bus I²C exclusivo (PS-IIC 0 o PS-IIC 1) para prevenir colisiones (ambas cámaras utilizan la dirección 0x3C). Los controladores opera a 100 kHz en modo *polling*. Las señales PCLK / HSYNC / VSYNC / D[7:0] alimentan directamente el bloque *Video In to AXI4-Stream,* en el cual la lógica de la sección PL del dispositivo ZYNQ genera un *frame done* que posteriormente detectan las rutinas *ping-pong* de los *buffers*, por lo que no se requiere de la gestión de interrupciones en la sección PS. Las líneas RST_N y PWDN asociadas a GPIO 54 y 55 respectivamente se controlan desde el *driver* que asegura una secuencia de *power-up* reproducible.

4.7.2.5. Secuencia final de gestión de las cámaras.

La gestión de las cámaras OV5640 en la aplicación software se estructura en una secuencia final que asegura la posibilidad de obtener un proceso de inicialización reproducible y un flujo de vídeo continuo:

1. Inicialización del sensor: En su código interno, la función *OV5640_Init()* ejecuta la secuencia temporal recomendada por el fabricante: mantiene la señal RST_N a nivel

bajo y la señal PWDN a nivel alto, aplica DOVDD/AVDD, libera primero la señal de PWDN, luego RST_N y espera la disponibilidad del bus SCCB para empezar a programar los registros de configuración.

2. Verificación de integridad: La función *main()* realiza la lectura del registro 0x3100 de cada sensor OV5640, se compara con 0x78, garantizando así la correcta conexión y funcionamiento antes de continuar con el flujo de vídeo.
3. Carga de configuración SCCB: La rutina *OV5640_Config()* ejecuta la lectura de *ov5640_setting_VGA_640_480_YUV422[]* y *ov5640_setting_HD_1280_720_YUV422[]* aplicando secuencialmente los pares registro–datos necesarios para configurar: el modo de vídeo, el PLL interno, el *binning* 2×2 y la ventana de lectura. Para finalizar, libera el bit de *stand-by* con los registros 0x3008=0x02 para que ambos sensores puedan comenzar a generan un flujo activo de datos.
4. Comienzo de captura continua: Tras liberar la señal de *reset* y esperar la estabilización del bus SCCB, el bloque de vídeo detecta la primera señal de *frame_done* y activa los *buffers ping-pong*, iniciando la captura y escritura continua en memoria sin más intervención de la sección PS.

Con esta secuencia las cámaras se inician en un estado conocido y generan vídeo de forma estable.

4.7.3. Gestión de los buffers

Para almacenar y transferir las imágenes de las cámaras de forma eficiente, la aplicación hace uso de los núcleos IP *Video Frame Buffer Write* y *Video Frame Buffer Read* en un esquema que comprende un doble *buffer* (*ping-pong*). En definitiva, se asignan dos regiones de memoria DDR para cada cámara: de manera que mientras una región de *buffer* envía la información de un *frame* para el *display*, la segunda región de *buffer* va escribiendo de manera simultánea el *frame* siguiente que se ha capturado, almacenados así en el *buffer* regiones de memoria alternadas. Gracias a esta implementación se garantiza que el flujo de datos es constante y, al mismo tiempo, se evita ocasionar *tearing* de imagen, ya que no se puede leer y escribir en el mismo espacio de memoria al mismo tiempo. En esta implementación, se definen direcciones

fijas en DDR para los *buffers* correspondientes a cada cámara (Código 4), correspondientes a regiones separadas en la memoria donde se encuentran almacenados los *frames* de las cámaras de manera alternada.

Código 4. Definiciones e instancias framebuffers

```
#define BUFFER0_ADDR1 (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x0A000000)
#define BUFFER1_ADDR1 (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x0B000000)
#define BUFFER0_ADDR2 (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x0C000000)
#define BUFFER1_ADDR2 (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x0D000000)

// Separate buffer tracking for each camera
static u32 currentWriteBuffer1;
static u32 currentReadBuffer1;
static u32 currentWriteBuffer2;
static u32 currentReadBuffer2;

/*
 * Instancias globales de las IP de Write y Read
 */
XV_frmbufwr frmbufwr_inst1, frmbufwr_inst2;
XV_frmbufrd frmbufrd_inst1, frmbufrd_inst2;
```

La lógica del doble *buffer* está encapsulada en las funciones *FrameBufferStartDoubleBuffer()* y *FrameBufferCheckWriteComplete()* que se encuentran en el fichero *vframebuffers.c*.

Al iniciar la aplicación, tras la configuración de las cámaras, la aplicación SW hace llamadas a *FrameBufferStartDoubleBuffer1* y *FrameBufferStartDoubleBuffer2* para inicializar el esquema de doble *buffer* de cada una de las cámaras. La función *FrameBufferStartDoubleBuffer()* (Código 5) asigna inicialmente el *buffer* de lectura (*currentReadBuffer*) al espacio *BUFFER0_ADDR1* que va a ser usado para la salida de vídeo y el *buffer* de escritura (*currentWriteBuffer1*) al espacio *BUFFER1_ADDR1,* en el cual la cámara comenzará a volcar el nuevo *frame*. Internamente, esta función realiza la configuración del IP *Frame Buffer Read* apuntando a la dirección *currentReadBuffer1* y la inicia de modo que continuamente muestra el contenido de ese buffer. Al mismo tiempo, configura el IP *Frame Buffer Write* para que escriba en *currentWriteBuffer1* con la opción de auto-reinicio desactivada, activándola en modo *frame* a *frame*. Esto último implica que la IP de escritura capturará un *frame* completo en el *buffer* que ha sido asignado y quedará en modo *idle* hasta que sea reconfigurado, permitiendo así al *software* decidir el momento del intercambio de *buffers*.

Código 5. Rutinas FrameBufferStartDoubleBuffer2: gestión del doble buffer.

```
//CAM2
void FrameBufferStartDoubleBuffer2(VideoMode vMode)
{
    currentReadBuffer2  = BUFFER0_ADDR2;
    currentWriteBuffer2 = BUFFER1_ADDR2;

    FrameBufferReadSetup2(vMode, currentReadBuffer2);
    FrameBufferWriteSetup2(vMode, currentWriteBuffer2);

}
```

La función *FrameBufferCheckWriteComplete()* (Código 6) se invoca en cada iteración del bucle principal. Desde que detecta que la IP de escritura ha quedado en modo *idle*, comienza el intercambio *"ping-pong":* el *buffer* recién completado se convierte en fuente de lectura y el otro se convierte en escritura, actualizando inmediatamente las direcciones de ambos IP. La lectura nunca se para y la visualización sigue sin interrupciones ni *tearing*, con un flujo estable con tan solo dos *buffers*.

Código 6. FrameBufferCheckWriteComplete2: gestión del doble buffer

```
void FrameBufferCheckWriteComplete2(VideoMode vMode)
{
    // comprobamos si finalizó la IP de Write
    if (XV_frmbufwr_IsIdle(&frmbufwr_inst2)) {

        u32 completedBuffer = currentWriteBuffer2;

        u32 nextWriteBuffer = (completedBuffer == BUFFER0_ADDR2) ? BUFFER1_ADDR2 : BUFFER0_ADDR2;
        currentReadBuffer2 = completedBuffer;
        XV_frmbufrd_Set_HwReg_frm_buffer_V(&frmbufrd_inst2, currentReadBuffer2);

        currentWriteBuffer2 = nextWriteBuffer;

        FrameBufferWriteSetup2(vMode, currentWriteBuffer2);

    }
}
```

4.7.4. Gestión del IP *Video Mixer*

El IP *Video Mixer* gestiona las salidas de los dos flujos de datos *AXI-Stream* de las cámaras y su resultado final es un solo flujo HDMI. La inicialización de la parte del IP *Video Mixer* se encuentra encapsulado en las funciones *VMixerInitialize()* y *VMixerStart()* dentro del código del fichero *mixer.c* que se invocan a continuación desde la función *main()*. En la inicialización global (*VMixerInitialize*), se invoca a la función *XVMix_Initialize()* y a continuación se deshabilita la capa *master* con a *XVMix_LayerDisable(&vmix_inst, XVMIX_LAYER_MASTER).*

Se construye el objeto *XVidC_VideoStream* con los parámetros de salida como se muestran en el Código 7.

Código 7. Asignación de parámetros de la instancia XVidC_VideoStream

```
// Configurar el mixer para YUV 422
    XVMix_LayerDisable(&vmix_inst, XVMIX_LAYER_MASTER);

    VidStream.VmId = XVIDC_VM_1280x720_30_P;
    VidStream.ColorFormatId = XVIDC_CSF_YCRCB_422;
    VidStream.ColorDepth = XVIDC_BPC_8;
    VidStream.PixPerClk = XVIDC_PPC_1;
    XVidC_VideoTiming const *TimingPtr;
    TimingPtr = XVidC_GetTimingInfo(VidStream.VmId);
    VidStream.Timing = *TimingPtr;
    VidStream.FrameRate = XVidC_GetFrameRate(VidStream.VmId);

    XVMix_SetVidStream(&vmix_inst, &VidStream);
```

Esta estructura se aplica al IP por medio de la función *XVMix_SetVidStream()* y a continuación se reactiva la *master layer* llamando a *XVMix_LayerEnable(),* quedando como fondo neutro sin origen asignado, y se inicia el mezclador mediante la llamada a la función *XVMix_Start()*. En la configuración de capas (*VMixerStart*) se definen dos ventanas (*XVidC_VideoWindow*) que dividen el *frame* (Código 8). Esto coloca la imagen asociada a *layer 1* a la mitad izquierda y la de *layer 2* en la mitad derecha de la pantalla.

Código 8. Definición de MixLayerConfig: posición y tamaño de las capas

```
int VMixerStart()
{
    static const XVidC_VideoWindow MixLayerConfig[2] =
    {//  X    Y    W    H
        {0,   0, 640, 720},//LAYER 1
        {640, 0, 640, 720} //LAYER 2
    };
```

Para cada capa se calcula el *stride*: $2\ bytes \times 640\ px = 1280\ bytes/linea$, y se aplica mediante la función *XVMix_SetLayerWindow()* (Código 9). En el hardware se fija *alpha* al valor 256 mediante *XVMix_SetLayerAlpha()*, asegurando opacidad total.

Código 9. Configuración de ventana, stride y alfa de cada capa

```
for (layerIndex=XVMIX_LAYER_1; layerIndex<XVMix_GetNumLayers(&vmix_inst); ++layerIndex) {
    Win = MixLayerConfig[layerIndex-1];

    XVMix_GetLayerColorFormat(&vmix_inst, layerIndex, &Cfmt);
    xil_printf("   Layer Color Format: %s\r\n", XVidC_GetColorFormatStr(Cfmt));
    Stride = 2; //BytesPerPixel YUV Format
    Stride *= Win.Width;

    xil_printf("   Set Layer Window (%3d, %3d, %3d, %3d): ",
            Win.StartX, Win.StartY, Win.Width, Win.Height);
    Status = XVMix_SetLayerWindow(&vmix_inst, layerIndex, &Win, Stride);
    if(Status != XST_SUCCESS) {
    xil_printf("ERROR Command XVMix_SetLayerWindow Failed \r\n");
    } else {
        xil_printf("DONE\r\n");
    }

    xil_printf("   Set Layer Alpha to %d: ", XVMIX_ALPHA_MAX);
    if (XVMix_IsAlphaEnabled(&vmix_inst, layerIndex)) {
        Status = XVMix_SetLayerAlpha(&vmix_inst, layerIndex, XVMIX_ALPHA_MAX);
```

Por último, cada capa se activa mediante la llamada a la función *XVMix_LayerEnable()*. El resultado es un flujo 720p compuesto en el que las dos imágenes se muestran al mismo tiempo, con la capa maestra desplazada al fondo inactivo.

4.8. PRUEBAS DE VÍDEO EN TIEMPO REAL

En este apartado se comprueba en tiempo real la cadena captura-procesamiento-visualización de la plataforma de prueba expuesta en los epígrafes anteriores, comprobando además si las dos cámaras OV5640, junto a la salida HDMI, funcionan sin interrupciones (sin pérdida de fotogramas ni efecto de *tearing*). Para ello la metodología final de funcionamiento de la aplicación es la siguiente:

1. En el inicio de la aplicación, se programa el *bitstream* y se lanza el archivo ELF desde la UART.
2. Al iniciarse se comprueba el ID 0x3100 = 0x78 de cada uno de los dos sensores OV5640, en caso de que la lectura no sea correcta, se aborta la prueba correspondiente.
3. Se activa el doble buffer con la llamada a las funciones *FrameBufferStartDoubleBuffer1() y FrameBufferStartDoubleBuffer2()* que reservan dos regiones de DDR por cámara e inician la ejecución de los IP en modo doble *buffer*.

4. Para la inicialización de salida de video se habilita el *Test Pattern Generator* (TPG) estableciendo la resolución, y se llaman a las funciones *DisplayInitialize()* y *DisplayStart()* que inicia la visualización de vídeo a través de HDMI a 1280 x 720 p60.
5. El IP *Video Mixer* habilita dos capas de tamaño 640 x 720 px y las ubica en la mitad izquierda y derecha del cuadro, quedando *master layer* como fondo neutro.
6. En el bucle principal, la rutina *FrameBufferCheckWriteComplete1/2()* se encarga del proceso de intercambio de direcciones en cada VSYNC, asegura que no se opere nunca sobre la misma región de la memoria DDR.
7. Finalmente, la salida HDMI muestra ambas cámaras en paralelo.

En la Figura 67, se puede observar cómo hay un ligero *offset* hacia la izquierda. El centro óptico del sensor no coincide exactamente con el centro geométrico de la lente, por lo que el contenido se proyecta ligeramente desplazado.

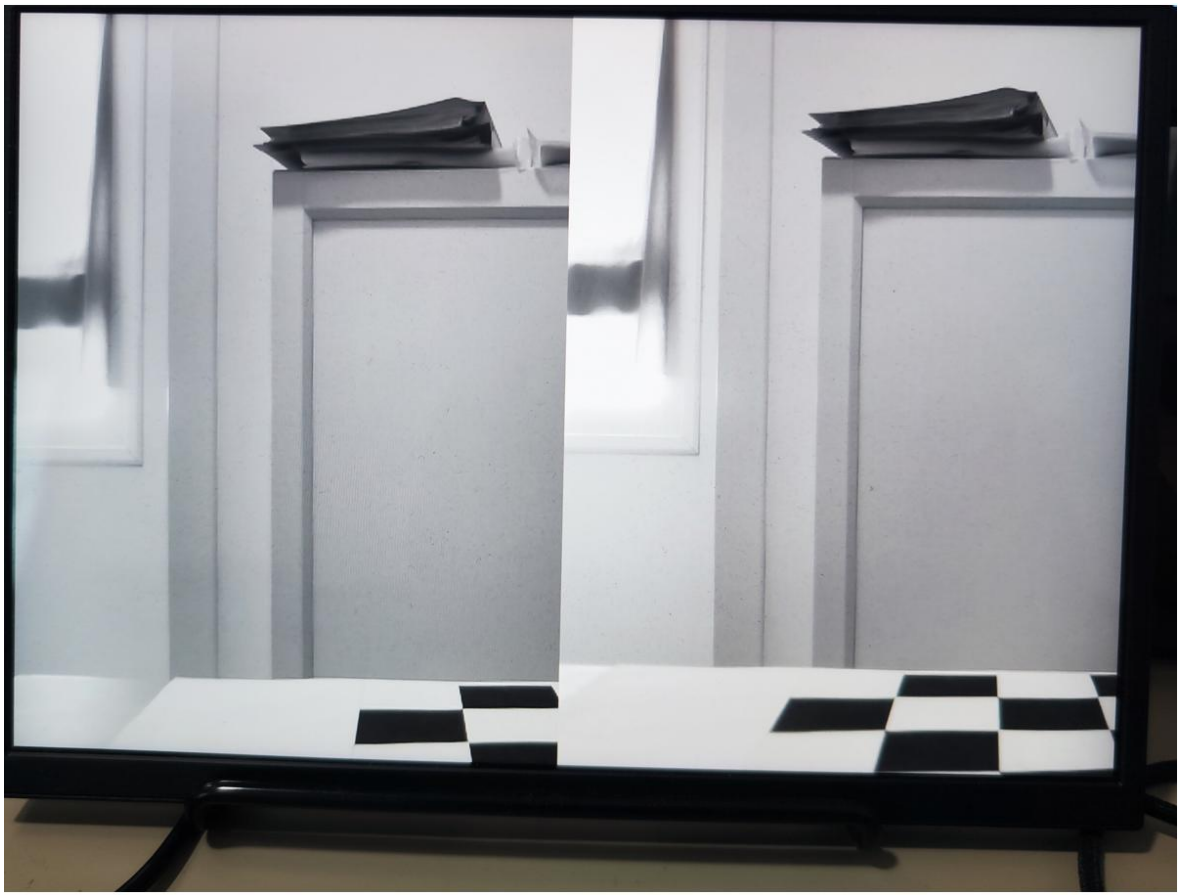

Figura 67. Visualización lado a lado sobre el monitor HDMI

Las pruebas realizadas verifican que la plataforma inicial cumple con los requisitos de captura y visualización en tiempo real y que está lista para la integración del acelerador SLBM que permitirá la generación de mapas de profundidad.

CAPÍTULO 5. IMPLEMENTACIÓN DEL ALGORITMO SLBM CON VITIS VISION

En este capítulo, se presenta el algoritmo Stereo Local Block Matching (SLBM) elegido para el cálculo de la disparidad y su adaptación hardware a través de Vitis HLS y de la biblioteca Vitis Vision. Se expone la arquitectura del acelerador hardware, el banco de pruebas, las fases de simulación, síntesis y co-simulación, así como su integración como núcleo IP AXI-Stream en la cadena de vídeo de la plataforma implementada en la placa de prototipado PYNQ-Z2.

5.1. INTRODUCCIÓN AL ALGORITMO SLBM (STEREO LOCAL BLOCK MATCHING)

Stereo Local Bolck Matching (SLBM) [36] es un algoritmo de correspondencia estéreo local y fundamentado en la técnica de *Block Matching*, para el cálculo de horizontes de correspondencia de disparidad a partir de un par de imágenes estéreo rectificadas. El objetivo es obtener las correspondencias entre la imagen de la parte izquierda y la de la derecha correlacionando pequeñas regiones de la imagen denominadas "bloques". La diferencia que existe en la posición del conjunto de la imagen de ambos lados se denomina disparidad y se encuentra relacionada inversamente con la distancia al objeto. Existe una gran variedad de funciones de coste [37] que pueden emplearse como criterio de coincidencia. Las más habituales son: la *suma de diferencias absolutas* (SAD), *correlación cruzada normalizada* (NCC), *transformada de rango* (RT) y *transformada census* (CT) [38]. En concreto, la función de coste BM $\hat{C}$ para un bloque vecino N_p alrededor de cada píxel p se expresa como:

$$\hat{C}(pd) = \sum_{q \epsilon\, N_p} C(q, d), \tag{5}$$

con la función de coste por pixel *C:*

$$C(pd) = \Theta\big(I_1(p),\, I_2(p\, +\, d)\big), \tag{6}$$

donde p es la posición del píxel en la primera imagen I_1, d es la disparidad, mientras que I_2 representa la segunda imagen y Θ representa la métrica de coste a nivel de píxel. El promediado de los costes de píxel C para obtener el coste de bloque $\hat{C}$ introduce un problema conocido como *fattening*. Esto significa que los valores de disparidad en el mapa tienden a difuminarse. En la mayoría de los casos, las disparidades del primer plano se extienden sobre las del fondo; por ello este efecto se denomina a veces *foreground-fattening* y también se conoce como *bleeding* [39].

La razón por la que se elige la técnica SLBM para este TFG se debe a que se presenta como un término medio entre la complejidad computacional y la capacidad para su paralelización. En comparación con técnicas más complejas (p.e. *Semi-Global Matching* o técnicas usando optimización global [40]) las técnicas de basadas en BM local presentan una complejidad computacional limitada, como puede ser el tamaño de ventana y el rango de disparidades, lo que permite implementaciones eficientes en *hardware* (Figura 68). La técnica *Block Matching* local, es muy popular en el ámbito del *hardware* programable ya que su carácter local y regular permite que se pueda implementar en arquitecturas paralelas sobre FPGA. Aunque los métodos globales pueden conseguir una mayor exactitud en los mapas de disparidad, suelen requerir para ello más recursos de cómputo y de memoria. SLBM en cambio se puede beneficiar de estructuras fuertemente paralelizadas, procesando antes diferentes píxeles o disparidades al mismo tiempo para conseguir unas prestaciones en tiempo real, lo que representa una ventaja para un SoPC basado en FPGA.

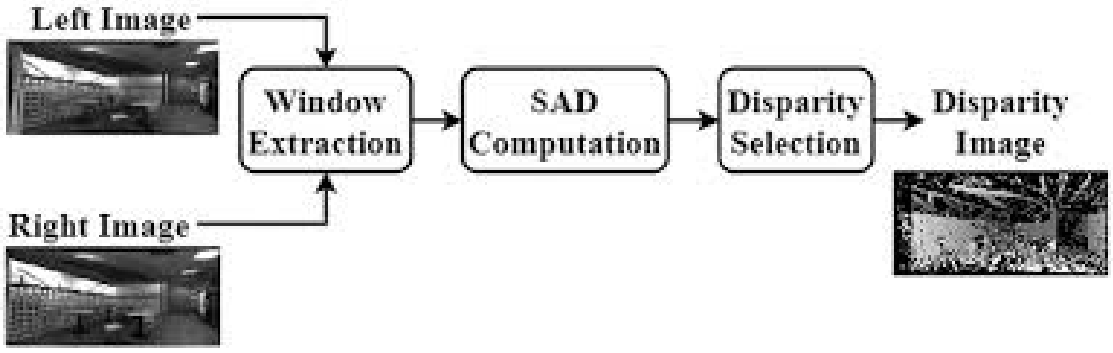

Figura 68. Arquitectura simplificada de la aplicación [36]

5.1.1. Entorno de desarrollo: Vitis HLS, biblioteca Vitis Vision y OpenCV

El cálculo *hardware* de la disparidad requiere de una cadena de herramientas que permita traducir la lógica de alto nivel a puertas lógicas en la FPGA sin que el diseñador necesite realizar su descripción en lenguaje VHDL o Verilog. Para ello, AMD/Vitis proporciona un entorno que permite realizar el proceso HLS (*High-Level Synthesis*). Este proceso toma funciones que están descritas en C/C++ (o en OpenCL C) y las transforma, usando el compilador v++, en RTL optimizado. Se aplican directivas de paralelismo tales como *pipeline*, *unroll* o *dataflow*, que pueden incluirse como directivas *pragmas HLS*, para, finalmente, empaquetar el resultado como un núcleo IP listo para su uso en el entorno Vivado IP Integrator.

En la práctica, el flujo se basa en cuatro fases automatizables (*csim*, *csynth*, *cosim* y *export_design*) todas invocables desde un único *script* Tcl, adecuado para una rápida iteración. Durante la síntesis, la herramienta informa de los recursos necesarios para su implementación en términos de LUT, FF, DSP, BRAM, así como latencia estimada, lo que permite establecer la frecuencia objetivo antes de pasar a la implementación.

Por otro lado, OpenCV (*Open Source Computer Vision Library*) [41] es un proyecto abierto creado por Intel que contiene más de 2.500 algoritmos optimizados en lenguaje C/C++ para el aprendizaje automático y la visión artificial. Sus módulos abarcan desde operaciones de procesamiento de imágenes básicas (*imgproc*) hasta calibración y reconstrucción en 3D (*calib3d*), entrada/salida de video (*videoio*) y visualización (*highgui*). La licencia es *BSD 3-Clause* que permite la integración sin restricciones en proyectos tanto académicos como comerciales. También cuenta con enlaces oficiales para Python, Java, y MATLAB y aceleración mediante CUDA y OpenCL en plataformas compatibles.

A su vez, AMD posee la biblioteca Vitis Vision. Una librería pública de plantillas *hardware* que emulan la funcionalidad de muchas de las rutinas clásicas de OpenCV (como, StereoBM, remap, etc.) pero descritas en C++ y preparadas para su uso en HLS. Sus primitivas son invocadas en el contenedor propio *xf::cv::Mat* y puertos *AXI4-Stream* a RAM interna, de forma que basta cambiar el prefijo *cv::* por *xf::cv::* y añadir los *pragmas* para obtener una aceleración HW en la sección PL del dispositivo ZYNQ.

Por tanto, la concurrencia con OpenCV es completa:

- Prototipos y casos de referencia: Primero se desarrollan y depuran en OpenCV, sobre el *host* (ARM Cortex-A9), para obtener resultados de referencia y generar los casos de prueba.
- Migración a Vitis Vision. Las mismas funciones se reemplazan por sus homónimas en *xf::cv*, se añaden *pragmas* y se fijan los parámetros *compile-time* (profundidad de *pixel*, NPPC, rango de disparidades...).
- Comparativa en *testbench*. En la fase de co-simulación, la salida del modelo RTL se compara bit a bit con la obtenida en OpenCV para asegurarse de que la traducción no introduce errores numéricos. El resultado es un acelerador replicable y portable: la parte computacionalmente intensiva reside en la FPGA, mientras que OpenCV permanece en el procesador para usarlo en calibración, visualización o post-procesado *offline*.

Finalmente, el algoritmo SLBM se implementa como un núcleo *hardware* generado con Vitis HLS que hace uso de la primitiva *xf::cv::StereoBM* de la biblioteca Vitis Vision, aprovechando su compatibilidad con OpenCV para reutilizar código y casos de prueba. Esta decisión permite concentrar la carga computacional de la correspondencia estéreo en la lógica programable del dispositivo Zynq-7000, manteniendo un flujo de diseño homogéneo y fácilmente escalable.

Cabe mencionar que inicialmente en esta fase no se ha llevado a cabo ni la calibración, ni la rectificación de las cámaras OV5640 (se implementan en el Capítulo 6), por tanto, las imágenes que se han usado para realizar las pruebas iniciales del algoritmo provienen de conjuntos de imágenes estándar y no de las cámaras del sistema. De este modo es posible comprobar la correcta ejecución del bloque del cálculo de disparidad en su modo aislado.

5.2. DESARROLLO EN C++/HLS CON VITIS VISION

En este apartado se desarrolla un bloque IP para implementar el algoritmo SLBM mediante Vitis HLS 2023.1, aprovechando las funciones proporcionadas por la biblioteca Vitis Vision. En particular, se ha utilizado la función *xf::cv::StereoBM* de dicha biblioteca, la cual implementa el algoritmo *block matching* local. Se parte de la base del proyecto de referencia

stereolbm que se encuentra dentro de los ejemplos de Vitis Vision, el cual se adapta para cumplir los requisitos específicos de la aplicación desarrollada en este TFG.

En dicho ejemplo, se puede observar el código utilizado para poder implementar el acelerador (*xf_stereolbm_accel.cpp*) que contiene la lógica que permite obtener la disparidad, mientras que el *testbench* (*xf_stereolbm_tb.cpp*) se encarga de realizar la lectura de las imágenes y llamar al núcleo HLS. Las imágenes son representadas con el contenedor estándar *xf::cv::Mat* que se obtiene mediante las funciones de conversión entre *AXI4-Stream* y *xf::cv::Mat*.

En el entorno de desarrollo AMD Vitis, todo *kernel* diseñado en C/C++ u OpenCL ha de ser convertido a lógica RTL (*Register-Transfer Level*), porque tiene que compilarse en la sección programable del dispositivo. El compilador *v++* se encargará de invocar el proceso *Vitis High-Level Synthesis* (HLS), que es el responsable de la generación del código RTL a partir del código fuente del *kernel*. Sin embargo, HLS también contiene un conjunto de *pragmas* que se incluirá en el propio código del *kernel* para ajustar el diseño a las necesidades de la plataforma en la que se integrará con interfaz *AXI4-Stream* [42].

5.2.1. Acelerador *stereolbm*

El acelerador describe el *pipeline hardware* StereoLBM, diseñado para calcular la profundidad a partir de un par de imágenes estéreo. Como se puede ver en la Figura 69, se comienza recibiendo dos flujos de vídeo *AXI4-Stream*, correspondientes a las imágenes de las vistas izquierda y derecha del array, las cuales se convierten a un formato de matriz interna para poder ser procesadas. En el siguiente paso, el núcleo del algoritmo (*xf::cv::StereoBM*), hace uso de las matrices obtenidas previamente en conjunto con unos parámetros preasignados como el umbral de textura, para generar un mapa de disparidad inicial en formato de 16 bits. Este formato, en el cual reside la disparidad inicial, no es un formato visible como tal, de manera que existe una etapa de post-procesado que hace el trabajo de escalar y normalizar los datos de un mapa de disparidad de 16 bits a una imagen estándar de 8 bits en escala de grises. Finalmente, la matriz de 8 bits que representa el mapa de disparidad final es empaquetada en un flujo de vídeo *AXI4-Stream* para su salida.

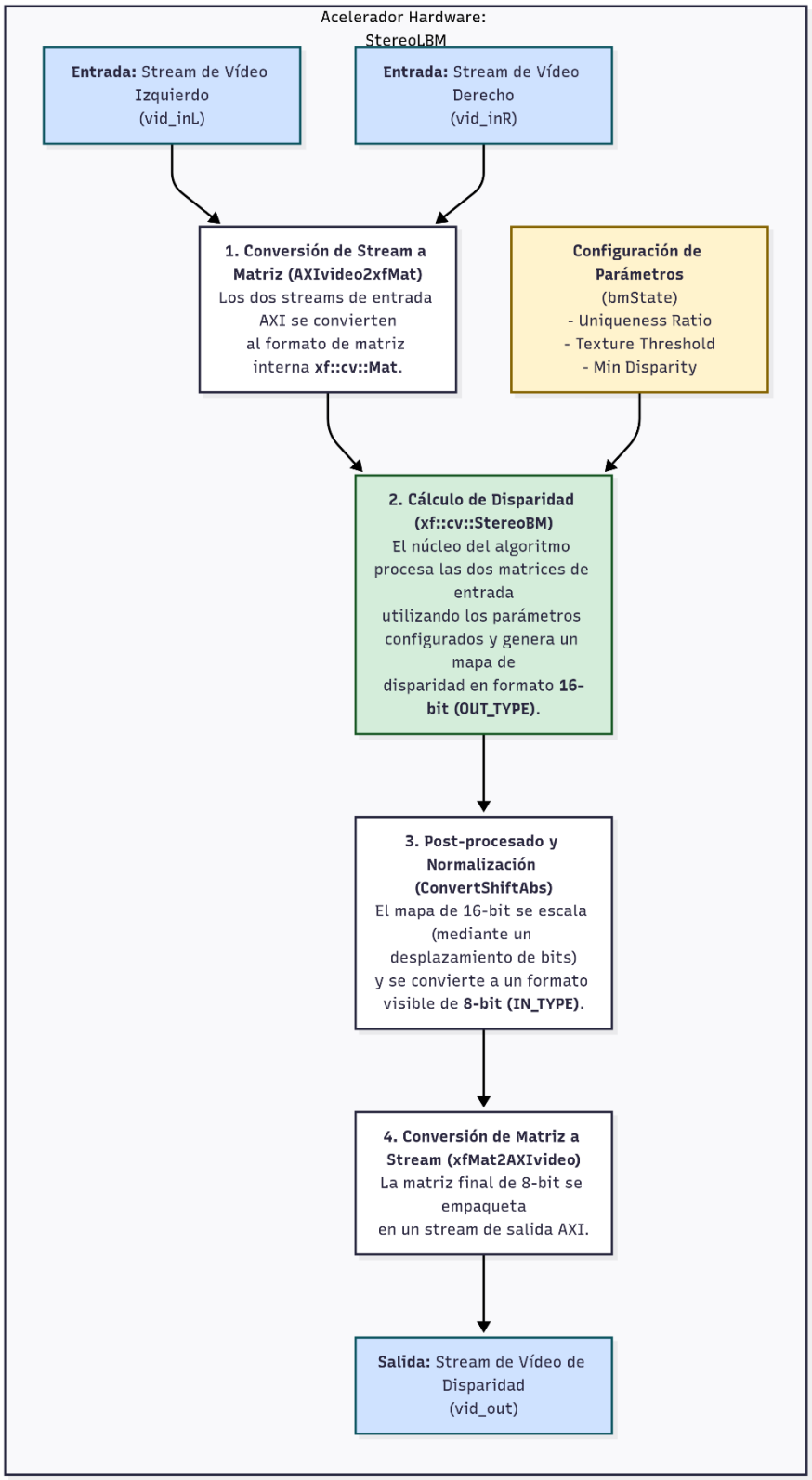

Figura 69. Diagrama de Flujo de la Pipeline de Datos del Acelerador StereoLBM

En el archivo *xf_stereolbm_accel.cpp*, se observa cómo se define la función *stereolbm_axis()*, cuyas interfaces AXI están declaradas como *pragmas* HLS.

Código 10. Declaración de las interfaces AXI mediante pragmas HLS

```
#pragma HLS INTERFACE axis port=vid_inL
#pragma HLS INTERFACE axis port=vid_inR
#pragma HLS INTERFACE axis port=vid_out
#pragma HLS INTERFACE s_axilite port=rows bundle=Ctrl
#pragma HLS INTERFACE s_axilite port=cols bundle=Ctrl
#pragma HLS INTERFACE s_axilite port=return bundle=Ctrl
```

El Código 10 muestra cómo la configuración de los puertos *vid_inL, vid_inR* y *vid_out* como interfaces *AXI4-Stream* con señales de tipo *TDATA, TVALID* y *TREADY*. Por otra parte, los puertos de control (*rows, cols* y *return*) quedarán como buses *AXI4-Lite*. En el cuerpo de la función se instancian matrices *xf::cv::Mat* para las imágenes de entrada y salida utilizando el *template* con parámetros como: profundidad de píxeles de entrada y salida, la altura y ancho de la imagen, el número de píxeles procesados por ciclo (NPPC) y la profundidad de la imagen (Código 11). Estos están definidos En el fichero *xf_config_params.h*.

Código 11. Declaración de matrices de imagen y estructura de estado xFSBMState

```
xf::cv::Mat<IN_TYPE, MAX_HEIGHT, MAX_WIDTH, NPPCX, XF_CV_DEPTH_IN_L> imgL_in(rows,cols);
xf::cv::Mat<IN_TYPE, MAX_HEIGHT, MAX_WIDTH, NPPCX, XF_CV_DEPTH_IN_R> imgR_in(rows,cols);
xf::cv::Mat<OUT_TYPE,MAX_HEIGHT,MAX_WIDTH,NPPCX,XF_CV_DEPTH_OUT> img_disp16u(rows,cols);

xf::cv::xFSBMState<SAD_WINDOW_SIZE, NO_OF_DISPARITIES, PARALLEL_UNITS> bmState;
```

A continuación, se inicializa el estado *bmState* del algoritmo *StereoBM* utilizando parámetros como `_PRE_FILTER_CAP_`, `_UNIQUENESS_RATIO_`, `_TEXTURE_THRESHOLD_` y `_MIN_DISP_`. Se utiliza la directiva `#pragma HLS DATAFLOW` para habilitar el paralelismo de flujo de datos y así procesar cada una de las etapas como una tarea independiente. Los principales parámetros de configuración vienen dados desde el fichero *xf_config_params.h*. Tras un análisis experimental de estos parámetros, se ajustan a los siguientes valores:

- `SAD_WINDOW_SIZE = 15`, que establece el tamaño de la ventana de búsqueda de bloques;
- `NO_OF_DISPARITIES = 128`, que corresponde en este caso al rango máximo de disparidades a calcular;
- `PARALLEL_UNITS = 16`, que es la cantidad de unidades de procesamiento en paralelo (NPPC) para acelerar el cálculo.

- Las macros como `_TEXTURE_THRESHOLD_`, `_UNIQUENESS_RATIO_`, `_PRE_FILTER_CAP_` y `_MIN_DISP_` se inicializan en principio a los valores estándar 20, 15, 31 y 0, respectivamente que limitan los filtros y el rango mínimo de disparidad.
- Se define `SHIFT (short)(log10((NO_OF_DISPARITIES*16.0)/256.0)/log10(2))`, como el factor de escalado que se utiliza para convertir el valor de la disparidad de 16 bits a 8 bits.

La función *stereolbm_axis()* es la que lleva a cabo la tarea principal en estas etapas. En primer lugar, se realiza la conversión de los flujos *AXI4-Stream* de entrada, en matrices *xf::cv::Mat* a través de *xf::cv::AXIvideo2xfMat(vidX_in, imgX_in)*. Una vez convertidos los datos, se hace una llamada al *kernel* de *StereoBM* de Vitis Vision como función *template* (Código 12).

Código 12. Conversión de flujos y llamada al kernel StereoBM

```
// Retrieve xf::Mat objects from img_in data:
xf::cv::AXIvideo2xfMat(vid_inL, imgL_in);
xf::cv::AXIvideo2xfMat(vid_inR, imgR_in);

// Run xfOpenCV kernel:
xf::cv::StereoBM<SAD_WINDOW_SIZE, NO_OF_DISPARITIES, PARALLEL_UNITS, IN_TYPE, OUT_TYPE,
MAX_HEIGHT, MAX_WIDTH, NPPCX, XF_USE_URAM, XF_CV_DEPTH_IN_L, XF_CV_DEPTH_IN_R,
XF_CV_DEPTH_OUT>(imgL_in, imgR_in, img_disp16u, bmState);

xf::cv::Mat<IN_TYPE,MAX_HEIGHT,MAX_WIDTH,NPPCX,XF_CV_DEPTH_OUT> img_disp8u(rows, cols);

ConvertShiftAbs(img_disp16u, img_disp8u);

// Convert _dst xf::Mat object to output array:
xf::cv::xfMat2AXIvideo(img_disp8u, vid_out);
```

Esto lleva a cabo el *Block Matching* local que genera una matriz de disparidad de 16 bits llamada *img_disp16u*. Para convertirla en un mapa de disparidad de 8 bits, se llama a la función *ConvertShiftAbs()*, que recorre píxel a píxel la imagen de disparidad 16U, desplazando la imagen a la derecha usando el operador SHIFT predefinido, y tomando el valor absoluto en 8 bits.

Para finalizar, la matriz resultante de 8 bits (*img_disp8u*) se convierte nuevamente en un flujo *AXI4-Stream* con *xf::cv::xfMat2AXIvideo(img_disp8u, vid_out)*, que envía la imagen de disparidad al exterior como una secuencia de vídeo. Así, el flujo de datos en el *kernel* avanza de la siguiente manera: los datos procedentes de los puertos *AXI4-Stream* pasan a matrices *xf::cv::Mat*, a partir de ahí son procesados por el algoritmo *StereoBM* más la conversión de bits

y, finalmente, salen de nuevo por *AXI4-Stream*. El resultado final permite que el diseño pueda integrarse perfectamente en Vivado como un bloque IP con puertos *AXI4-Stream* de video, permitiendo su integración en el flujo con los otros bloques IP.

5.2.2. *Testbench*

Para comprobar que la funcionalidad del IP de disparidad que se genera a partir de HLS es correcta, se realiza una co-simulación entre el código C++ (OpenCV) y la función HLS. En este caso, el archivo de *testbench* (*xf_stereolbm_tb.cpp*) que realmente actúa como aplicación host, utiliza OpenCV solo para leer las imágenes de prueba y calcular una referencia de disparidad. Después convierte estos datos a flujos AXI y llama al *kernel* HLS, simulando así el diseño. Como indica el manual de Vitis Vision, el *testbench* invoca al acelerador desde un archivo C++ separado, mientras que el acelerador implementa las funciones de visión [43].

El diagrama de flujo de la Figura 70 describe el proceso de validación del *kernel hardware* StereoLBM. El *testbench* comienza cargando un par de imágenes estéreo (izquierda y derecha) que servirán como datos de entrada. A partir de este punto, la ejecución se bifurca para crear dos resultados paralelos: por un lado, se procesan las imágenes con la función *cv::StereoBM* de la librería OpenCV para generar una salida de referencia o "*golden*"; simultáneamente, las mismas imágenes se envían al acelerador *hardware* (DUT - *Device Under Test*) para obtener su resultado. Una vez que ambos procesos finalizan, el flujo converge en la etapa de comparación, donde se calcula la diferencia píxel a píxel entre la imagen de referencia y la generada por el *hardware*. Finalmente, se analiza esta diferencia y se emite un veredicto: si el error es cero, el *test* se declara superado, validando que el acelerador es funcionalmente correcto; en caso contrario, el *test* falla.

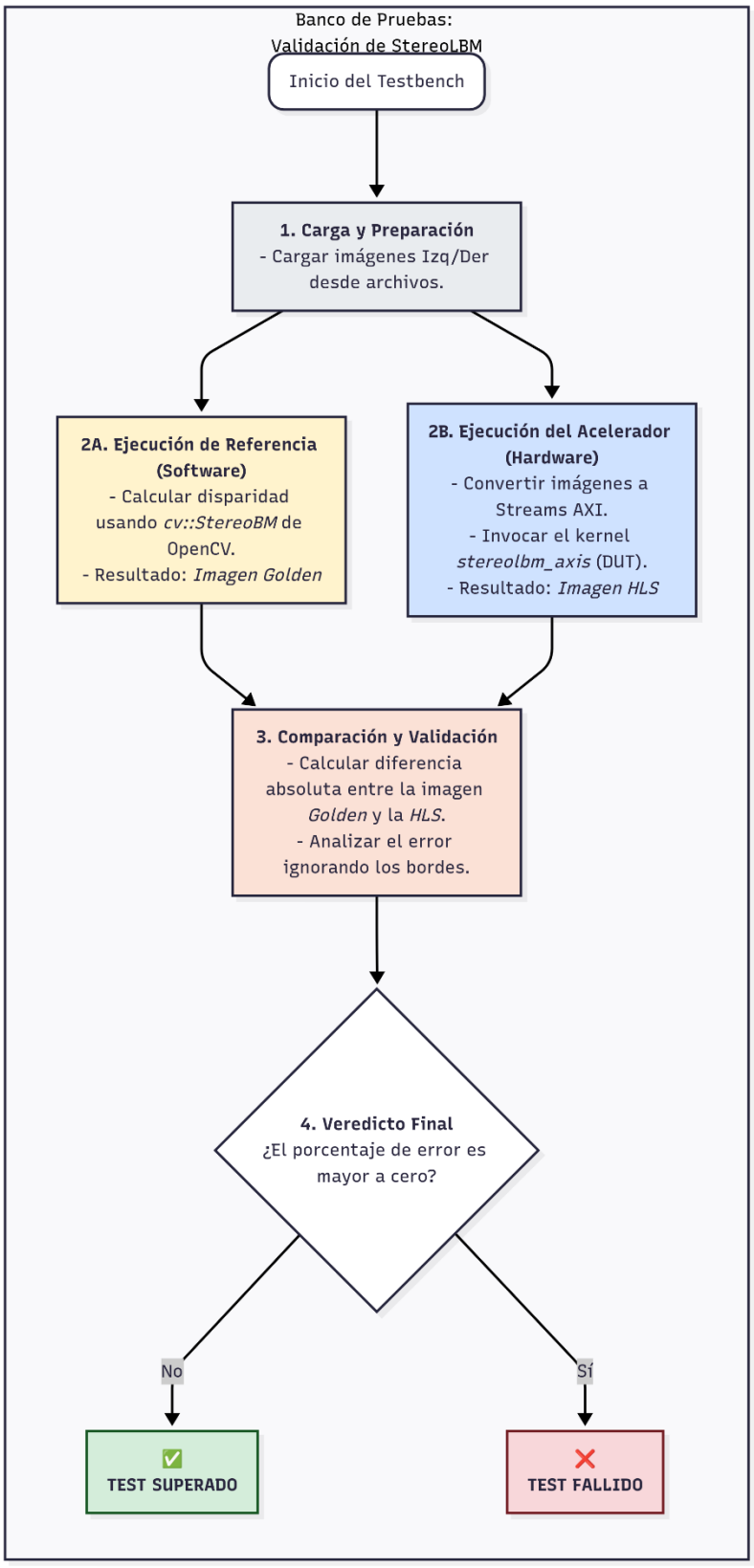

Figura 70. Flujo de Validación del Acelerador en el Banco de Pruebas

El *testbench* desarrollado realiza los siguentes pasos:

1. Calcular la referencia OpenCV.

El programa arranca cargando el par estéreo en escala de grises con *cv::imread*, para a continuación calcular el mapa de disparidad con *cv::StereoBM*. La conversión a 8 bits y el volcado de la imagen de referencia (ocv_output.jpg) se muestran en el Código 13**.**

Código 13. Lectura de imágenes y generación de la referencia OpenCV

```
// ./xf_stereolbm_tb.cpp (extracto)
if (argc != 3) { fprintf(stderr,"Usage: %s <L> <R>\n",argv[0]); return EXIT_FAILURE; }
cv::Mat left_img  = cv::imread(argv[1], 0);
cv::Mat right_img = cv::imread(argv[2], 0);
cv::Ptr<cv::StereoBM> sbm = cv::StereoBM::create(NO_OF_DISPARITIES, SAD_WINDOW_SIZE);
sbm->setPreFilterCap(_PRE_FILTER_CAP_);
sbm->setTextureThreshold(_TEXTURE_THRESHOLD_);
sbm->setUniquenessRatio(_UNIQUENESS_RATIO_);
sbm->compute(left_img, right_img, disp16);
disp16.convertTo(disp8, CV_8U, (256.0/NO_OF_DISPARITIES)/16.);
cv::imwrite("../../../../src/data/ocv_output.jpg", disp8);
```

2. Preparación de los flujos AXI y ejecución del *kernel*.

Las dos imágenes se empaquetan como los flujos de datos *AXI4-Stream* mediante la función *xf::cv::cvMat2AXIvideoxf*, se lanzan al núcleo HLS *stereolbm_axis*, y la salida vuelve a transformarse a cv::Mat. Este proceso queda recogido en el Código 14.

Código 14. Conversión a AXI4-Stream y llamada a stereolbm_axis

```
hls::stream<ap_axiu<8,1,1,1>> vid_inL, vid_inR, vid_out;
xf::cv::cvMat2AXIvideoxf<NPPCX,8>(left_img,  vid_inL);
xf::cv::cvMat2AXIvideoxf<NPPCX,8>(right_img, vid_inR);
stereolbm_axis(vid_inL, vid_inR, vid_out, rows, cols);
xf::cv::AXIvideo2cvMatxf<NPPCX>(vid_out, out_img);
cv::imwrite("../../../../src/data/hls_out.jpg", out_img);
```

3. Comparación de resultados.

Se calcula la diferencia absoluta píxel-a-píxel; antes de analizarla se descarta un borde de `SAD_WINDOW_SIZE` para evitar artefactos. En la función *xf::cv::analyzeDiff*, si el porcentaje

de error es cero el test pasa, confirmando que la salida HLS coincide funcionalmente con la referencia OpenCV (Código 15).

Código 15. Análisis de la diferencia y validación del resultado

```
cv::absdiff(disp8, out_img, diff);
cv::Rect roi(SAD_WINDOW_SIZE, SAD_WINDOW_SIZE,
             diff.cols - (SAD_WINDOW_SIZE<<1),
             diff.rows - (SAD_WINDOW_SIZE<<1));
float err_per;  xf::cv::analyzeDiff(diff(roi), 1, err_per);
if (err_per > 0.0f) { fprintf(stderr,"ERROR: Test Failed\n"); return 1; }
else                { std::cout << "Test Passed" << std::endl; }
```

Todas las constantes críticas se comparten a través del fichero *xf_config_params.h*, garantizando una configuración idéntica en ambos dominios. Estas macros afectan a su vez la llamada a la función *cv::StereoBM*, así como también a la estructura interna *xFSBMState* utilizada dentro del *kernel* HDL. De esta forma, se puede asegurar que la comparación sea idéntica.

Con este *testbench* se consigue que la validación sea exhaustiva en la co-simulación, donde el flujo OpenCV establece la causa de la referencia «*golden*», mientras que el DUT entrega el *hardware*. El chequeo bit a bit que asegura *analyzeDiff* permite verificar que no haya discrepancias numéricas antes de entrar en las fases de *csynth*, *cosim* y el posterior empaquetado del IP, con lo que se minimizan riesgos en la integración del acelerador.

5.3. FLUJO DE GENERACIÓN Y VERIFICACIÓN DEL IP

Una vez validada la funcionalidad del *kernel* StereoLBM en lenguaje C/C++, el siguiente paso consiste en implementarlo como un núcleo IP sintetizable que pueda ser incluido dentro del diagrama de bloques de la arquitectura HW de la plataforma desarrollada en el entorno Vivado. Según AMD, en Vitis HLS el flujo de trabajo se distribuye en cuatro fases sucesivas: *csim*, *csynth*, *cosim* y *export_design*. Este proceso se puede automatizar completamente mediante un *script* Tcl sin hacer uso de la interfaz gráfica proporcionada. A través de la aplicación "*Vitis HLS 2023.1 command prompt*" se ejecuta dicho archivo Tcl (ver Código 16) que realiza el siguiente proceso:

5.3.1. Simulación en C (csim)

Este primer comando se encarga de compilar el testbench *xf_stereolbm_tb.cpp* y la implementación C/C++ del acelerador HLS, ejecutando ambos con el conjunto de imágenes de test y comparando los mapas de disparidad. El test genera tres ficheros (ocv_output.jpg, hls_out.jpg y diff_img.jpg) y aborta en caso de que el porcentaje de error sea distinto de cero. Tiene como objetivo el descarte de fallos lógicos que pudiesen perjudicar la síntesis *hardware*.

5.3.2. Síntesis en C (csynth)

Al invocar el comando *csynth_design,* el código C/C++ se convierte en el código RTL sintetizable, respetando las directivas de paralelismo definidas como *#pragma HLS*. La herramienta genera informes sobre la utilización de recursos LUT, FF, DSP, BRAM, así como las estimaciones sobre latencia. Para una frecuencia objetivo de 100 MHz, el diseño mantiene un píxel por ciclo de procesamiento.

5.3.3. Co-simulación C/RTL (cosim)

En la siguiente fase se ejecuta el comando `cosim_design -rtl` *verilog*. En este caso, Vitis HLS lanza el simulador *xsim* y compara la salida RTL contra la C, bit a bit. En los logs muestra "0 *errors*, 0 *warnings*" y se genera una traza VCD completa para la depuración. Este paso asegura que las transformaciones de la síntesis han sido respetadas y que no se han producido distorsiones numéricas que pudiesen perjudicar a la síntesis del diseño.

5.3.4. Exportar el diseño

Una vez se han superado las verificaciones, se empaqueta el núcleo como un bloque IP con su archivo *component.xml,* las vistas *xgui* y los ficheros con el código RTL sintetizable. Finalmente, el resultado se añade al IP Catalog de Vivado para la integración descrita en el epígrafe 5.4. Integración del *IP* stereolbm en la cadena de captura.

Código 16. Script TCL de automatización Vitis HLS para la generación y verificación del IP stereolbm

```
source settings.tcl

set PROJ "stereolbm.prj"
set SOLN "sol1"

if {![info exists CLKP]} {
  set CLKP 3.3
}

open_project -reset $PROJ

add_files "${XF_PROJ_ROOT}/L1/examples/stereolbm/xf_stereolbm_accel.cpp" -cflags " -I ${XF_PROJ_ROOT}/L1/examples/stereolbm/config -I${XF_PROJ_ROOT}/L1/include -I ./ -D__SDSVHLS__ -std=c++0x" -csimflags " -I ${XF_PROJ_ROOT}/L1/examples/stereolbm/config -I${XF_PROJ_ROOT}/L1/include -I ./ -D__SDSVHLS__ -std=c++0x"
add_files -tb "${XF_PROJ_ROOT}/L1/examples/stereolbm/xf_stereolbm_tb.cpp" -cflags " -I ${XF_PROJ_ROOT}/L1/examples/stereolbm/config -I${OPENCV_INCLUDE} -I${XF_PROJ_ROOT}/L1/include -I ./ -D__SDSVHLS__ -std=c++0x" -csimflags " -I ${XF_PROJ_ROOT}/L1/examples/stereolbm/config -I${XF_PROJ_ROOT}/L1/include -I ./ -D__SDSVHLS__ -std=c++0x"
set_top stereolbm_accel

open_solution -reset $SOLN

set_part $XPART
create_clock -period $CLKP

if {$CSIM == 1} {
  csim_design -ldflags "-L ${OPENCV_LIB} -lopencv_imgcodecs -lopencv_imgproc -lopencv_calib3d -lopencv_core -lopencv_highgui -lopencv_flann -lopencv_features2d" -argv " ${XF_PROJ_ROOT}/data/left.png ${XF_PROJ_ROOT}/data/right.png "
}

if {$CSYNTH == 1} {
  csynth_design
}

if {$COSIM == 1} {
  cosim_design -ldflags "-L ${OPENCV_LIB} -lopencv_imgcodecs -lopencv_imgproc -lopencv_calib3d -lopencv_core -lopencv_highgui -lopencv_flann -lopencv_features2d" -argv " ${XF_PROJ_ROOT}/data/left.png ${XF_PROJ_ROOT}/data/right.png "
}

if {$VIVADO_SYN == 1} {
  export_design -flow syn -rtl verilog
}

if {$VIVADO_IMPL == 1} {
  export_design -flow impl -rtl verilog
}

exit
```

5.4. INTEGRACIÓN DEL IP STEREOLBM EN LA CADENA DE CAPTURA

Una vez generado el IP mediante el proceso de síntesis de alto nivel, se exporta al diseño del entorno Vivado. Para ello, se añade al repositorio de IP permitiendo que Vivado lo encuentre dentro de su catálogo.

5.4.1. Incorporación en el diagrama de bloques

Dentro del diagrama de bloques del diseño de la arquitectura HW de la plataforma desarrollada, se añade el nuevo núcleo generado (Figura 71) y se comprueban los respectivos puertos de entrada de video *vid_inL* y *vid_inR* y la salida *vid_out* de 8 bits. Una vez verificado que el bloque tiene las interfaces *AXI4-Stream* y los puertos de control como buses *AXI4-Lite*, se procede a integrarlo en el *stream* de video. Para ello, simplemente se elimina el IP TPG de la plataforma inicial de prueba y se sustituye el *Video Mixer* por el nuevo IP (Figura 72). Por último, se vuelve a re-lanzar el proceso de síntesis, la implementación y la generación del *bitstream* para finalmente exportar la plataforma *hardware* lista para su integración en la aplicación *software*.

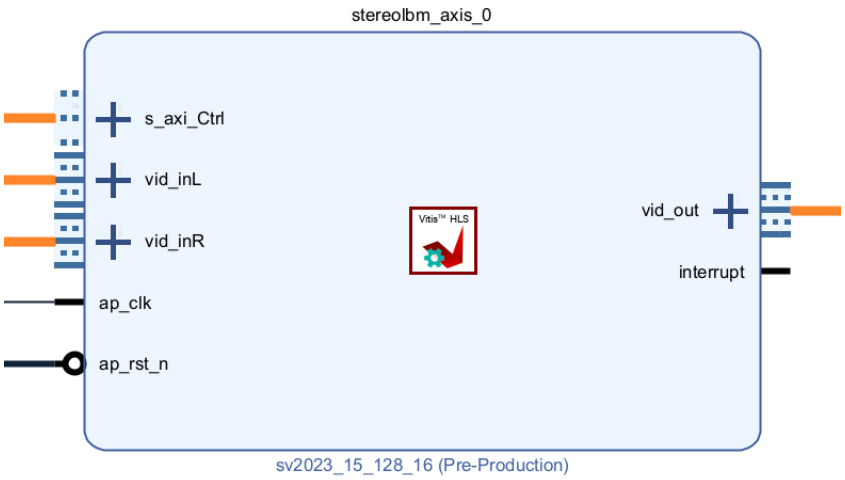

Figura 71. IP Stereolbm AXI-Stream

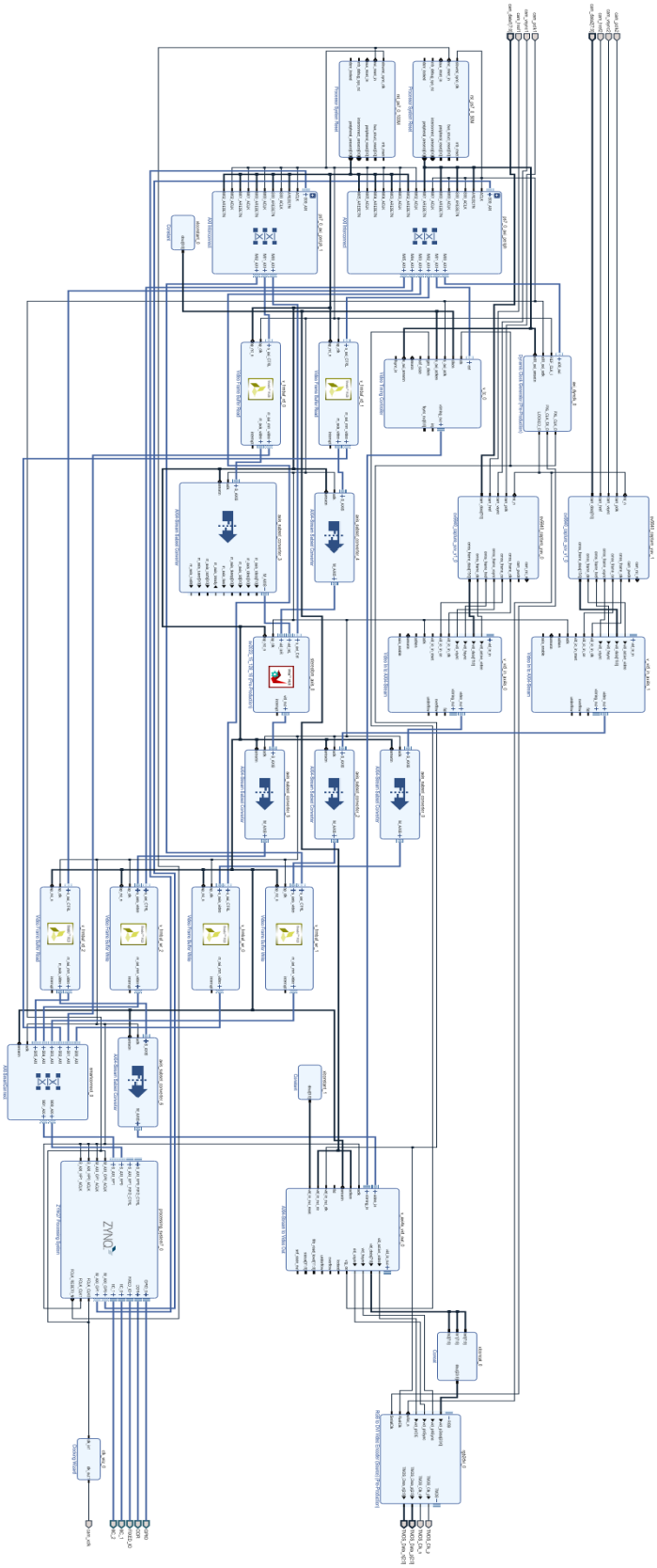

Figura 72. Diagrama de bloques de la arquitectura del sistema

5.4.2. Instancia a la API de *Stereo Vision*

Para mostrar el funcionamiento del IP SLBM, en el *software* de control, se implementa un módulo de abstracción en lenguaje C que encapsula la configuración y la inicialización del núcleo *hardware,* como se puede ver en el fichero *sv_ctrl.c* (Código 17). Se identifican en esta parte dos rutinas relevantes:

- *StereoVisionCtrl_init()*: recibe la configuración obtenida a partir del identificador del dispositivo de entrada (*device_id*), inicializa la instancia y marca el estado de preparación en el que se queda una vez ejecutada.
- *StereoVisionCtrl_setsize()*: determina las dimensiones de la imagen de entrada que se procesará en el bloque, utilizando las variables internas *rows* (número de filas) y *cols* (número de columnas).

Código 17. Inicialización y configuración de la IP Stereo LBM

```
#include "xstereolbm_axis.h"


int StereoVisionCtrl_init(XStereolbm_axis *InstancePtr, u16 device_id){
    int status;
    XStereolbm_axis_Config *config;
    config = XStereolbm_axis_LookupConfig(device_id);
    if (config == NULL) {
        InstancePtr->IsReady = 0;
        xil_printf("*E* Cannot get StereoVision lbm configuration\n");
        return XST_FAILURE;
    }


    status = XStereolbm_axis_CfgInitialize(InstancePtr, config);
    if (status != XST_SUCCESS) {
        xil_printf("*E* StereoVision lbm configuration initialization failed\n");
        return XST_FAILURE;
    }
    else {
        xil_printf("StereoVision lbm configuration initialization success\n");
    }
    return XST_SUCCESS;
}

int StereoVisionCtrl_setsize(XStereolbm_axis *InstancePtr, u32 width, u32 height){
    XStereolbm_axis_Set_rows(InstancePtr, height);
    XStereolbm_axis_Set_cols(InstancePtr, width);
    return XST_SUCCESS;
}
```

Para la incorporación de esta API en la aplicación principal *main.c* se realiza:

- Inicialización: se invoca a la función *StereoVisionCtrl_init()* con el identificador `STEREOVISION_ID` que está declarado en el fichero *xparameters.h*.
- Configuración del tamaño: la función *StereoVisionCtrl_setsize()* adapta el bloque al modo de vídeo (*vMode*) elegido.
- Verificación: lectura de líneas y de columnas mediante las funciones *XStereolbm_axis_Get_cols(&sv)* y *XStereolbm_axis_Get_cols(&sv)*
- Inicialización: habilitación del reinicio automático (*EnableAutoRestart*) y ejecución de la funcionalidad de la instancia (*Start*).

Con esta instancia, el núcleo SLBM se encuentra completamente operativo, soportando los flujos de datos proporcionados por los *framebuffers* en tiempo real y generando de forma continua el mapa de disparidad.

Finalmente, la Figura 73 y la Figura 74 muestran como referencia el mapa de disparidad en escala de grises obtenidos de un tablero de cuadros blancos y negros (damero que se ha utilizado posteriormente como patrón de calibrado). En ellos, se codifica en cada píxel la disparidad de la posición entre las dos vistas estereoscópicas. Las zonas con mayor disparidad (los puntos de la solución del tablero de cuadros más cercanos al sistema estéreo) son blancas, mientras que las que poseen una menor disparidad (los puntos de la solución del tablero de cuadros más alejados del sistema estéreo) se muestran en un tono de gris más oscuro. Aplicando este criterio, los cuadros que están más cercanos al sensor en el damero aparecen como los puntos que destacan como los más blancos (Figura 73) y, a la inversa, los cuadros que están más alejados van apareciendo en escalas de colores más oscuros (tonos grises o negro) (Figura 74), facilitando la interpretación visual y cuantitativa de la profundidad de la escena.

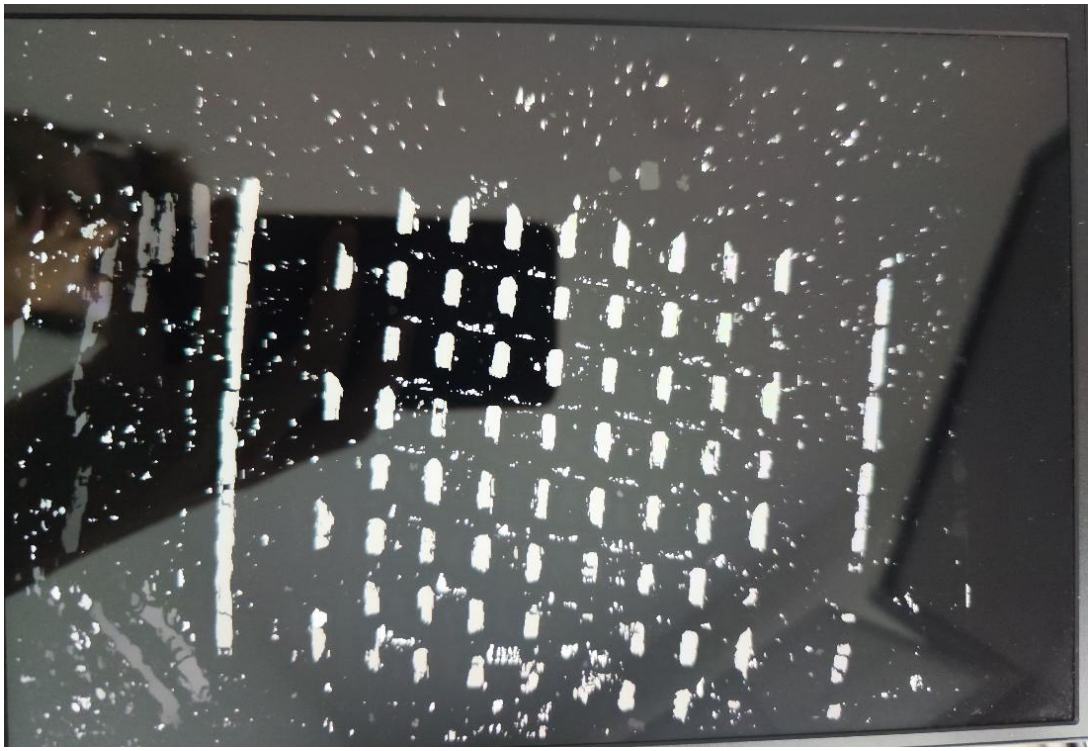

Figura 73. Mapa de disparidad — damero a corta distancia

Figura 74. Mapa de disparidad — damero a larga distancia

Como resultado, se puede ver el efecto bruto del núcleo IP HLS *Stereo* LBM cuando se procesan en paralelo los dos flujos de vídeo sin haber realizado corrección geométrica alguna, por tanto, el fondo queda prácticamente negro y sólo aparecen pequeños "puntos" dispersos de disparidad. Para paliar este aspecto, se realizan los procesos de calibración y rectificación estéreo ya que el interés de la plataforma desarrollada sólo se mostrará cuando se hayan incorporado estas etapas de preprocesado.

CAPÍTULO 6. INTEGRACIÓN Y GENERACIÓN DEL MAPA DE PROFUNDIDAD RECTIFICADO

En este capítulo se presenta la integración de todos los elementos para la obtención del mapa de profundidad rectificado. Se introduce el volcado de pares de imágenes capturadas a partir de la plataforma en una tarjeta micro-SD para realizar los procesos de calibración y rectificación offline. Adicionalmente, se aborda el cálculo de los parámetros intrínsecos y extrínsecos y la obtención de las tablas de remapeo. Finalmente, se expone la rectificación en tiempo real sobre FPGA, se incorpora el IP SLBM definitivo y se evalúa el rendimiento de la cadena completa sobre la plataforma.

6.1. ALMACENAMIENTO EN SD DE LA SECUENCIA DE IMÁGENES

Antes de poder pasar al proceso de calibración y rectificación *offline* de las cámaras, es necesario en primer lugar almacenar un número mínimo de pares de imágenes en una tarjeta microSD haciendo uso de los recursos de la placa de prototipado PYNQ-Z2 desde la aplicación *software*. Para ello se parte de la aplicación *software* realizada en la sección 4.7, y se introducen las modificaciones necesarias para almacenar correctamente los archivos RAW de ambos *buffers*. Así, dentro de un nuevo fichero (*sdcadrd_ctrl.c*) para la gestión del *dirver* SDIO, se han creado dos funciones denominadas *sdcard_init()* y *sdcard_transfer_write()* (Código 18) cuya funcionalidad es la inicialización y escritura en la SD.

- Función *sdcard_init()*: Se declara un objeto de tipo FRESULT (*file function return code*) y se hace uso de esta función para verificar que se monta FATFS (*Filesystem object structure*) en la partición 0:/ correctamente.
- Función *sdcard_transfer_write()*: Primero abre o crea el archivo especificado mediante la función *f_open()*, para verificar posteriormente que el puntero de la estructura del objeto de archivo esté posicionado al inicio, y así hacer la escritura bruta con la llamada a la función *f_write(),* que copia la longitud de los bytes desde la DDR indicada por la dirección del *buffer* de lectura actual. Finalmente se fuerza la sincronización de datos en la SD y se cierra el archivo mediante la función *f_close().*

Este es el flujo de trabajo estándar para inicializar, montar y escribir en la tarjeta SD.

Código 18. Funciones de inicio y escritura en la SD

```
#include "xil_types.h"
#include "ff.h"

int sdcard_init(void);
int sdcard_transfer_write(const char *file_name, u32 src_addr, u32 byte_len);
```

Finalmente, se hace un volcado en la SD dentro de la función *main()* del fichero *main.c*, donde se capturan en este caso 30 imágenes por cámara cada 1-2 segundos para dar tiempo a situar el damero de referencia en diferentes orientaciones con el objetivo de obtener una calibración posterior más precisa.

Durante la realización de este apartado se detectó inicialmente un *warning* que terminaba en error de compilación. Al incluir el *bitstream* de la plataforma, la librería *xilff* (*Generic Fat File System Library*) no se encontraba en el *path* del proyecto (Figura 75).

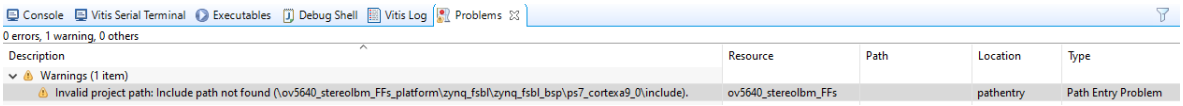

Figura 75.Warning Path Entry Problem

Para solucionar este error basta simplemente con incluir la librería *xilff* manualmente dentro de los drivers del *Board Support Package Settings* (Figura 76).

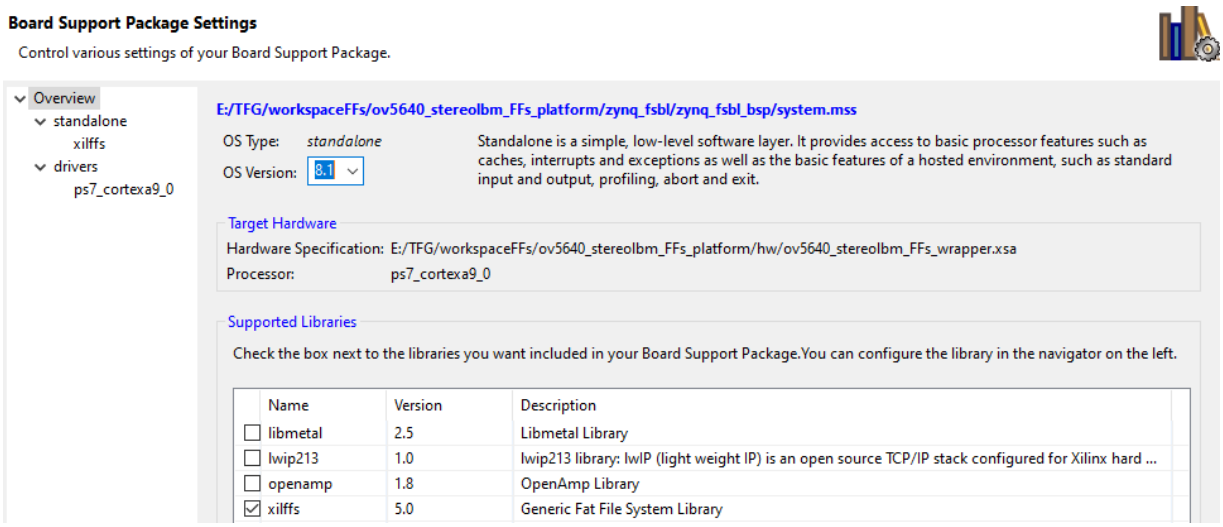

Figura 76. Ajustes del Board Support Package

Con este error corregido, se procede a capturar los 30 pares de imágenes de referencia de las cámaras en crudo (compresión de datos sin pérdidas).

El siguiente paso consiste en transformar los datos *RAW* en un formato que la aplicación de Matlab encargada del proceso de calibración de ambas cámaras pueda procesar. Para ello, se realiza una conversión mediante un *script* de Python que transforma los archivos binarios al tipo de archivo de imagen rasterizado PNG. El siguiente script (Código 19) de Python recorre todos los BIN de las dos cámaras, extrae la componente de luminancia (Y) y guarda cada *frame* en formato PNG en escala de grises. Una vez almacenadas las imágenes en este formato, se puede proceder a la fase de calibración *offline* en Matlab.

Código 19. Script en Python, Conversión RAW a PNG

```python
import os
import numpy as np
import matplotlib.pyplot as plt

# Parámetros de la imagen
WIDTH, HEIGHT = 800, 600
NUM_FRAMES   = 30

# Rutas a los directorios de BIN
LEFT_DIR  = os.path.join('imgs', 'leftcamera')
RIGHT_DIR = os.path.join('imgs', 'rightcamera')

def process_dir(src_dir, file_prefix, out_prefix):
    for i in range(NUM_FRAMES):
        suf    = f"{i:02d}"
        fname  = os.path.join(src_dir, f"{file_prefix}{suf}.BIN")
        outpng = f"{out_prefix}{suf}.png"

        if not os.path.isfile(fname):
            print(f"No existe {fname}, saltando.")
            continue

        # Lee RAW YUYV422
        raw = np.fromfile(fname, dtype=np.uint8)
        if raw.size != WIDTH * HEIGHT * 2:
            print(f" Tamaño inesperado en {fname}: {raw.size} bytes (esperado
{WIDTH*HEIGHT*2})")

        # Extrae sólo la luma (Y) cada 2 bytes
        Y = raw[0::2].reshape((HEIGHT, WIDTH))

        # Guarda en escala de grises
        plt.imsave(outpng, Y, cmap='gray', vmin=0, vmax=255)
        print(f"Guardado {outpng}")

if __name__ == "__main__":
    # Procesa la cámara izquierda
    process_dir(LEFT_DIR,  'CAM1_F',  'left_')
    # Procesa la cámara derecha
    process_dir(RIGHT_DIR, 'CAM2_F', 'right_')
    print("Conversión completa")
```

6.2. CALIBRACIÓN Y RECTIFICACIÓN DE LAS CÁMARAS

La calibración de una cámara consiste en estimar sus parámetros mediante imágenes de un patrón de calibración especial. Estos parámetros incluyen las características intrínsecas, los coeficientes de distorsión y las características extrínsecas de la cámara [44]. A su vez, la calibración estéreo trata de determinar la relación geométrica entre dos cámaras en el espacio. La manera de representar dicha relación es mediante la matriz de rotación *R* y el vector de traslación *T*. Por consiguiente, el principal objetivo de la calibración estéreo es la estimación de los elementos constituyentes de la matriz *R* y el vector *T* [45]. Para hallar estas matrices se realizan los siguientes pasos:

1. En primer lugar, se lleva a cabo la localización de correspondencias, donde se determina la posición de un mismo punto *P* en las imágenes capturadas por ambas cámaras.
2. Ese punto se pasa al sistema de coordenadas de la cámara izquierda o de la cámara derecha usando sus respectivas matrices de rotación R_0, R_1 y los respectivos vectores de traslación T_0, T_1:

$$P_0 = R_0 P + T_0, \qquad P_1 = R_1 P + T_1 \tag{7}$$

3. A partir de estas expresiones, se llega a $P_0 = RP_1 + T$ y sustituyendo se tiene:

$$(R_0 - RR_1)P + T_0 - RT_1 - T = 0 \tag{8}$$

4. Dado que *P* es arbitrario, se impone $R_0 - RR_1 = 0$ y $T_0 - RT_1 - T = 0$, lo que permite despejar *R* y *T*.
5. De esta manera, se llega a:

$$R = R_0 R_1^T, \qquad T = T_0 - RT_1 \tag{9}$$

6. Finalmente, calculando *R* y *T* para cada par de imágenes del patrón de calibración se obtienen diversas mediciones como el grado de coherencia, que determina la posibilidad de completar y eliminar el error en la estimación final de la geometría estéreo.

Por otro lado, el proceso de rectificación estéreo no solo elimina las distorsiones ópticas que cada cámara pudiera tener, sino que transforma estas dos imágenes para que se comporten como si hubieran sido capturadas por cámaras perfectamente coplanarias. El resultado es un par de imágenes cuyos pares epipolares quedan alineados, reduciendo así toda la búsqueda de correspondencias a un simple problema unidimensional y, por lo tanto, simplificando el cálculo de disparidad.

6.2.1. Introducción a la calibración *offline* mediante Matlab

Esta fase de calibración se llevará a cabo de manera *offline* utilizando Matlab. En ella se realiza la obtención de los parámetros intrínsecos y extrínsecos de cada cámara mencionados en el apartado anterior. Estos parámetros son fundamentales para las etapas posteriores de la rectificación de las imágenes y el cálculo del mapa de profundidad. Por ello, su correcta estimación es básica para el rendimiento del sistema.

Para realizar la calibración se utiliza la aplicación *Stereo Camerea Calibrator* de Matlab, que calcula los parámetros del par de cámaras a partir de un conjunto de imágenes de calibración. Se emplea un tablero de ajedrez como objeto de calibración debido a la precisión para estimar la geometría de las cámaras, ya que las esquinas de alto contraste del damero se pueden detectar con precisión *sub-píxel* y dan lugar a un conjunto regular y abundante de puntos de referencia que minimizan el error de reproyección. La Figura 77 muestra el damero de referencia usado para los procesos de calibración y rectificación.

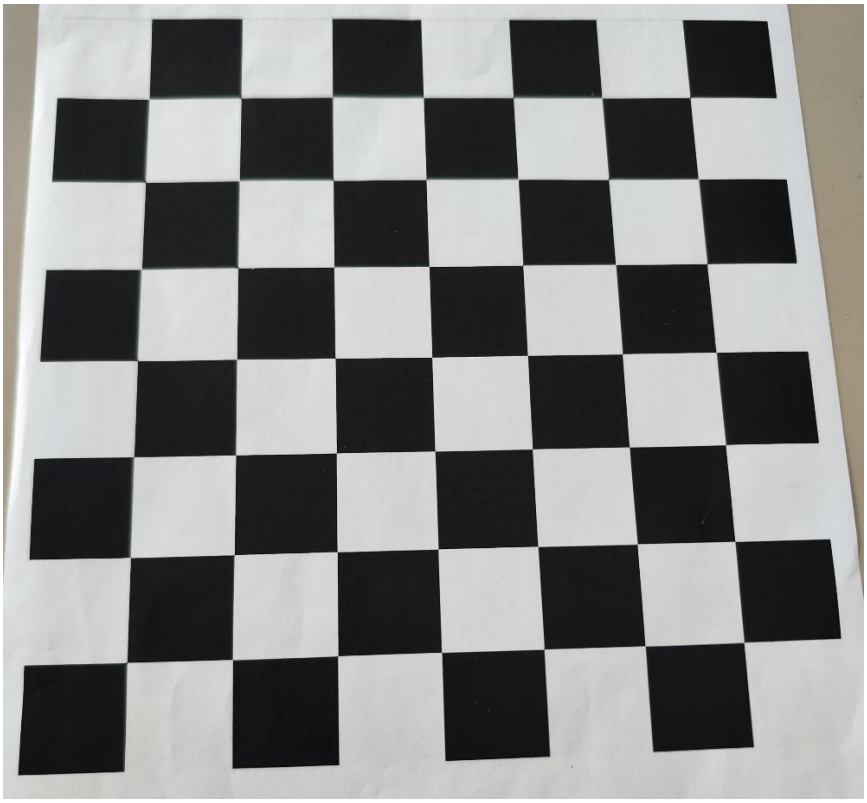

Figura 77. Damero de referencia usado

6.2.2. Uso de la aplicación *Stereo Camera Calibrator*

En este apartado se seguirá el flujo de trabajo explicado en el ejemplo de la documentación de la biblioteca *vision* de Matlab *Using the Stereo Camera Calibrator App* [46].

La aplicación *Stereo Camera Calibrator* permite calibrar lentes de cámara con un campo de visión (FOV) de hasta 95 grados, generando un objeto con los parámetros de la cámara estéreo. Este objeto se utilizará para calibrar las imágenes estéreo y exportar los parámetros. Dichos parámetros se exportan y se reutilizan en un *script* de *Python* utilizando OpenCV, en el que la función *cv2.stereoRectity()* aplica finalmente la matriz de rectificación a las cámaras.

El conjunto de funciones de calibración que utiliza la aplicación proporciona el flujo de trabajo para la calibración del sistema estéreo (Figura 78).

Figura 78. Flujo de trabajo para la calibración del sistema estéreo

6.2.2.1. Añadir pares de imágenes y seleccionar el modelo de la cámara

Una vez se tienen las imágenes preparadas para el proceso de calibración, se importaron como datos de entrada en la aplicación. La Figura 79 proporciona una imagen de la aplicación, donde se puede observar la interfaz correspondiente y varios pares de imágenes que contienen el damero de referencia en distintas posiciones. Después de la selección del patrón tipo *checkerboard* (dentro de la sección *Calibration Patterns*) y de las dimensiones del mismo, la aplicación analiza de forma automática la pareja de imágenes y trata de detectar el patrón de calibración en ambas.

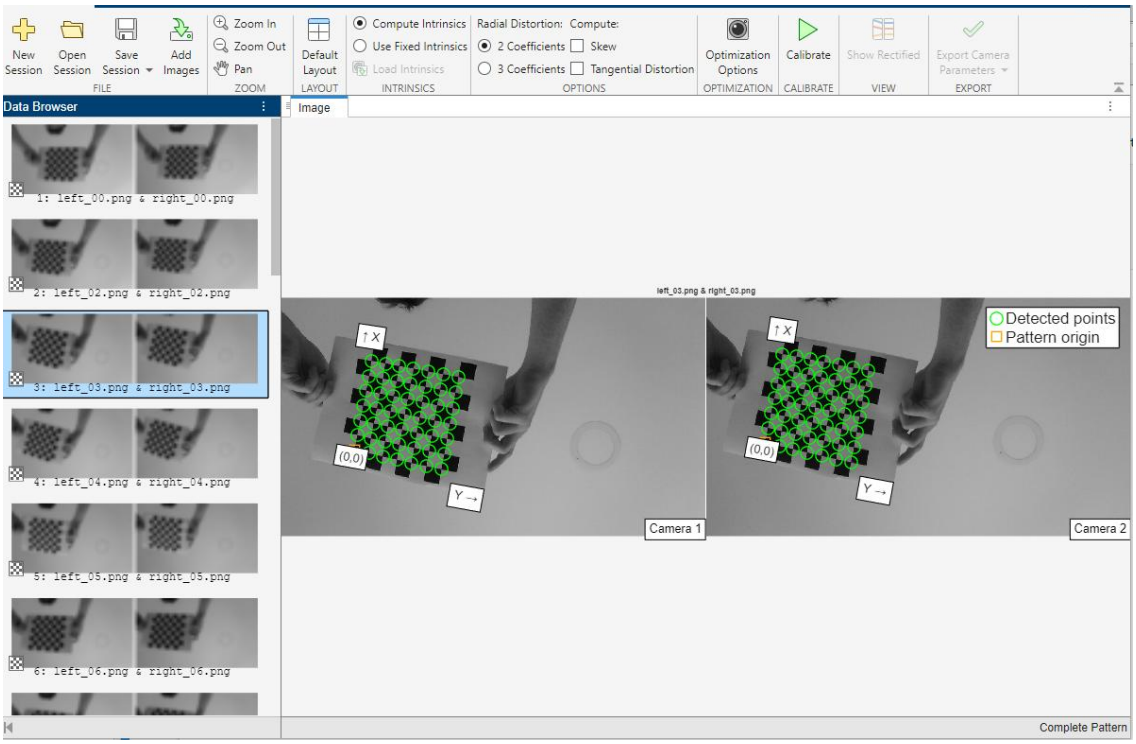

Figura 79. Pares de imágen en la aplicación Stereo Camera Calibrator

6.2.2.2. Calibrar las imágenes

Una vez seleccionada la opción de calibrar, la aplicación es capaz de determinar correctamente las esquinas internas de los cuadrados del tablero de referencia en todas las vistas, creando así un conjunto de puntos 2D coincidentes para cada par estéreo (Figura 80). Los puntos de esquina detectados constituyen la base para el cálculo de los parámetros de la cámara. Para incrementar la robustez en la calibración, se intentó que el conjunto de las imágenes incluyera las diversas orientaciones y posiciones en las cuales se podía colocar el tablero dentro del campo de visión de las cámaras. En concreto, se han utilizado unos 30 pares de imágenes, cumpliendo así las recomendaciones de variabilidad para una calibración correcta. Gracias a esta variabilidad de escena, el algoritmo dispone de más información sobre la distorsión de los lentes y la geometría estéreo, lo que se traduce en una estimación de los parámetros más exacta.

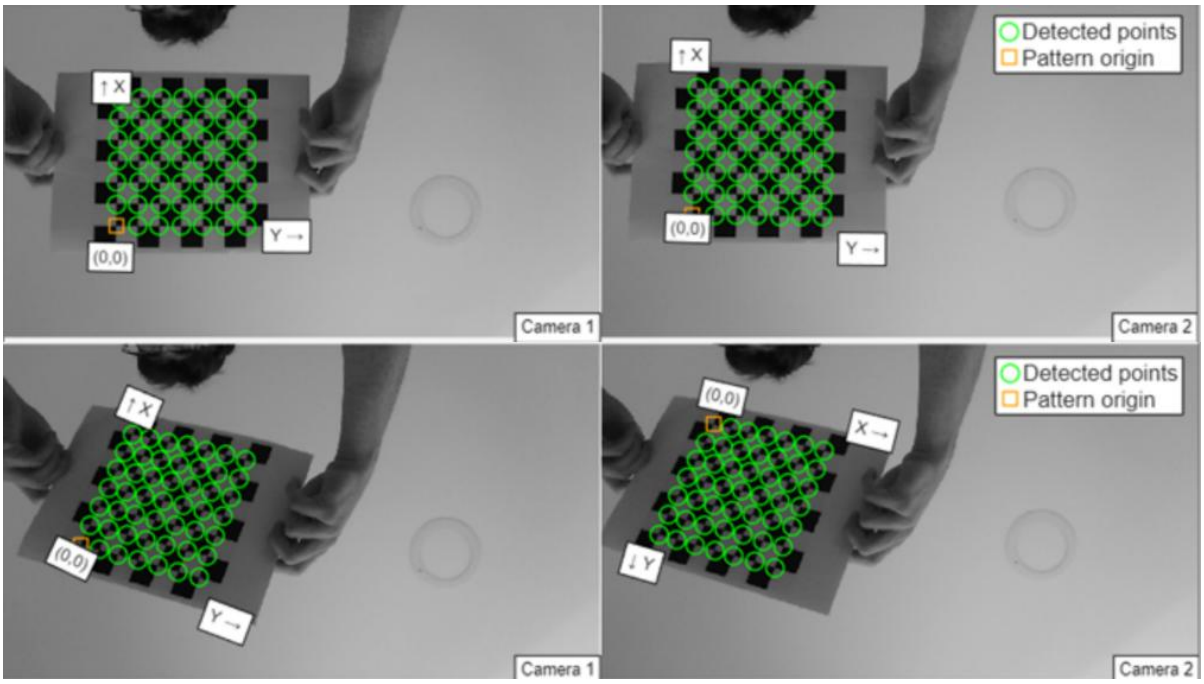

Figura 80. Diferencia entre los distintos pares de imágenes

6.2.2.3. Evaluar los resultados de la calibración y filtrado de outliers

Tras el proceso de calibración, la aplicación da como resultado un error de reproyección medio por imagen, que indica la calidad de la calibración obtenida. Con esto, se pueden localizar aquellas imágenes problemáticas cuya contribución empeora la robustez de la calibración. Algunas de estas parejas de imágenes presentan un error de reproyección mucho más alto que las demás, ya sea por detecciones de esquinas erróneas, o por geometrías del damero menos favorables. Para solucionar esto, la aplicación permite hacer filtrado de *outliers* con el fin de mejorar la respuesta. Una vez seleccionadas las que se consideran con un error de reproyección medio superior, se vuelve a ejecutar el cálculo del proceso de calibración (Figura 81). La recalibración posterior a la eliminación de los *outliers* produce un error relativamente mejorado, por tanto, se mejora la calidad de la calibración de los parámetros obtenidos.

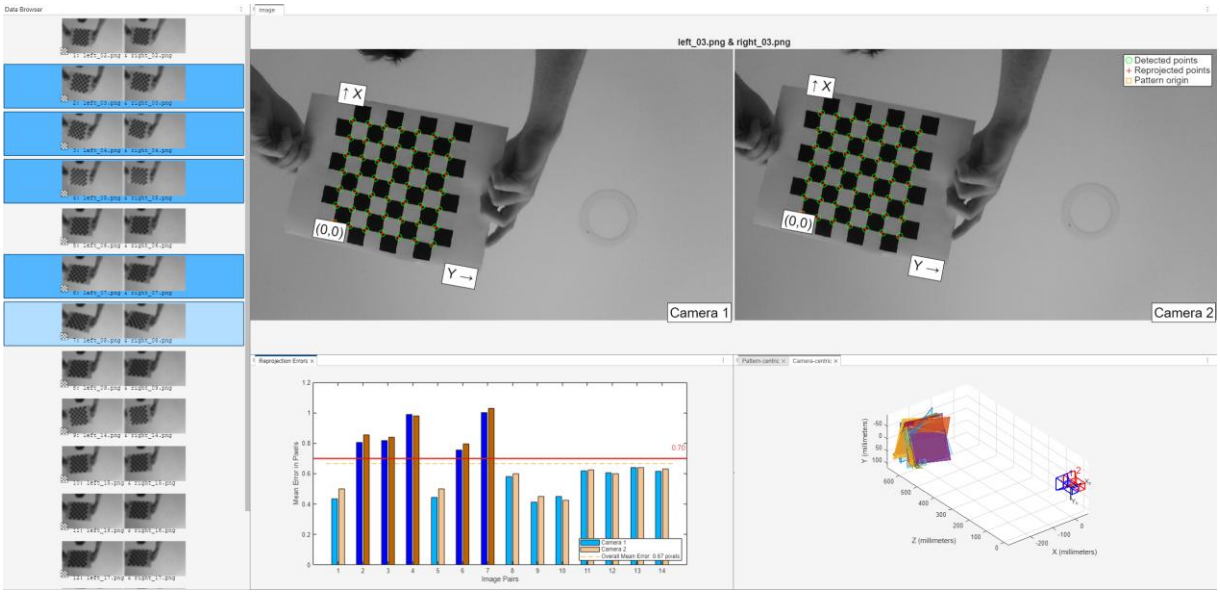

Figura 81. Filtrado de outliers y recalibración

Tras completar la calibración refinada, los resultados se verifican con las herramientas de visualización integradas en la aplicación. En la Figura 82 se muestra un ejemplo de las rectificaciones superpuestas a los pares de imágenes calibradas (opción *Show Rectified*). Se puede observar cómo las líneas horizontales aparecen alineadas entre las imágenes izquierda y derecha, confirmando así que la correción geométrica estéreo es válida tras la calibración, compartiendo un plano imagen rectificado común.

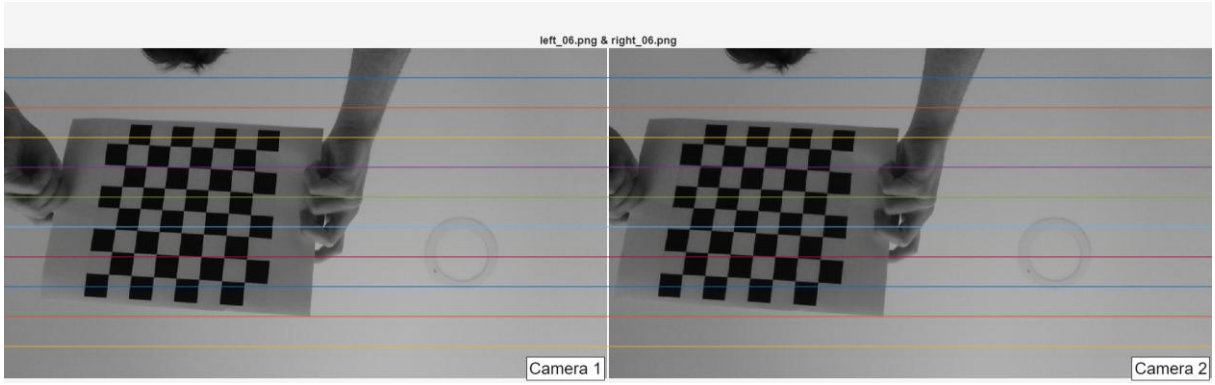

Figura 82. Líneas de rectificación (Show rectified)

6.2.2.4. Exportar parámetros de la cámara

Después del filtrado de los *outliers*, se puede comprobar que todos los errores se encuentran en un rango aceptable, como aparece en la Figura 83, lo que refuerza el hecho de que la calibración es coherente, con errores no excesivos en ninguno de los pares.

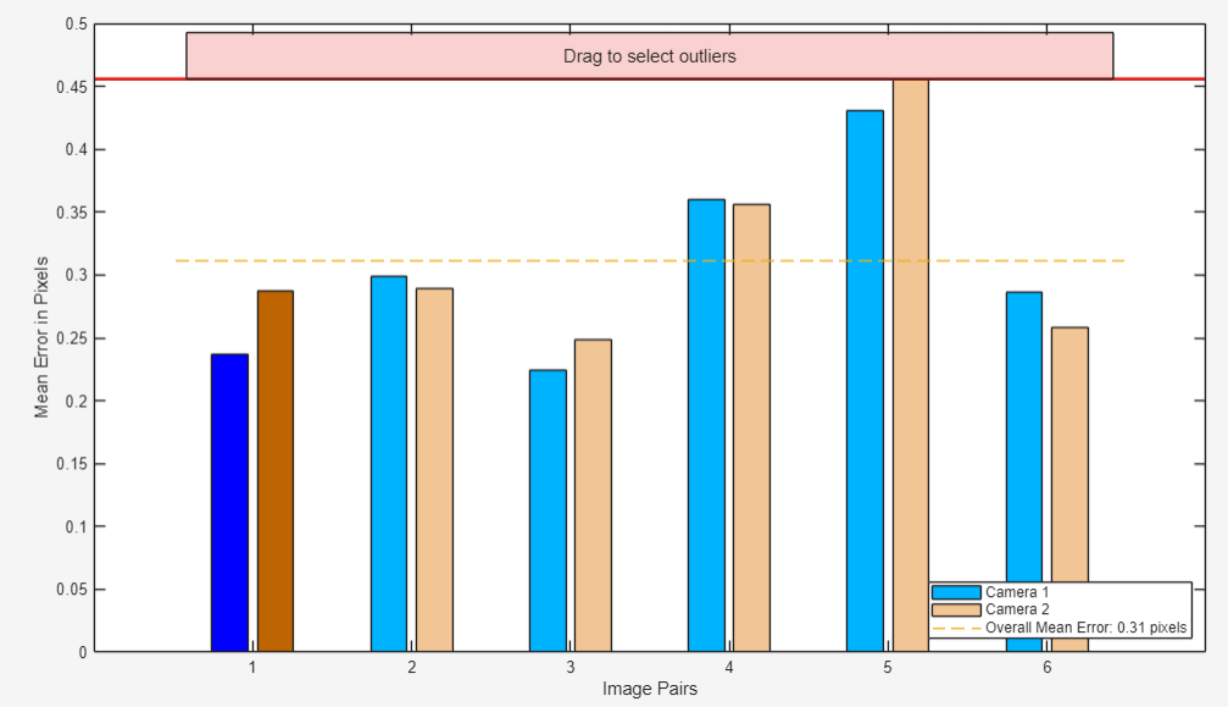

Figura 83. Filtrado final de errores de reproyección

Finalmente, la Figura 84 muestra la visualización 3D de los parámetros extrínsecos que se han obtenido (función *showExtrinsics*), donde se observa la posición y orientación relativas

de la cámara 2 respecto de la cámara 1 por la calibración. Finalmente, se exportan los parámetros intrínsecos y extrínsecos al *workspace* de Matlab para ser usados en etapas posteriores del proyecto, particularmente, en la rectificación de las secuencias capturadas por las cámaras del array en la plataforma *hardware*.

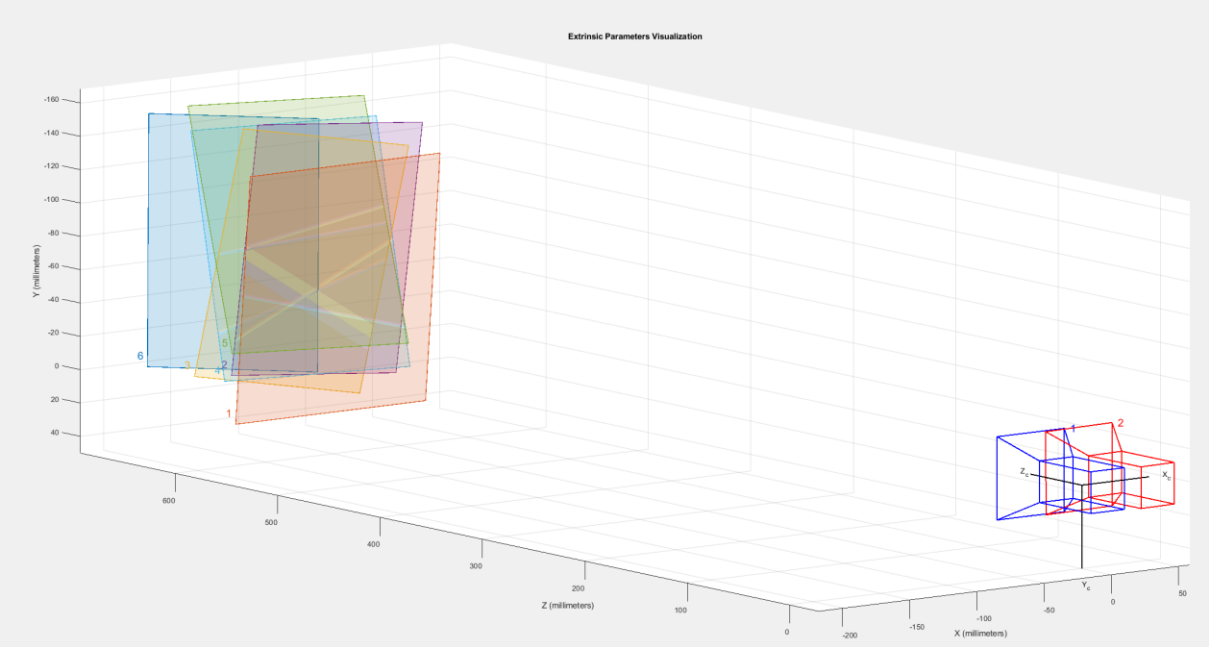

Figura 84. Visualización de los parámetros extrínsecos filtrados

6.2.4. Generación de parámetros intrínsecos y extrínsecos

Para comprobar que los parámetros se han exportado al *workspace* correctamente, Matlab genera un *script* automático que permite visualizar los extrínsecos asociados a la cámara 1 y los intrínsecos asociados a la cámara 2, así como los relativos a la posición y orientación de la cámara 2 sobre la 1(Figura 85). Los errores de estimación representan la incertidumbre de cada parámetro estimado. La función *estimateCameraParameters,* generada en el *script,* devuelve opcionalmente la salida *estimationErrors*, que contiene el error estándar correspondiente a cada parámetro de cámara estimado. El error estándar σ devuelto puede utilizarse para calcular intervalos de confianza. Por ejemplo, +/-1,96σ corresponde al intervalo de confianza del 95%. En otras palabras, la probabilidad de que el valor real de un parámetro dado se encuentre dentro de 1,96σ de su estimación es del 95 % [44].

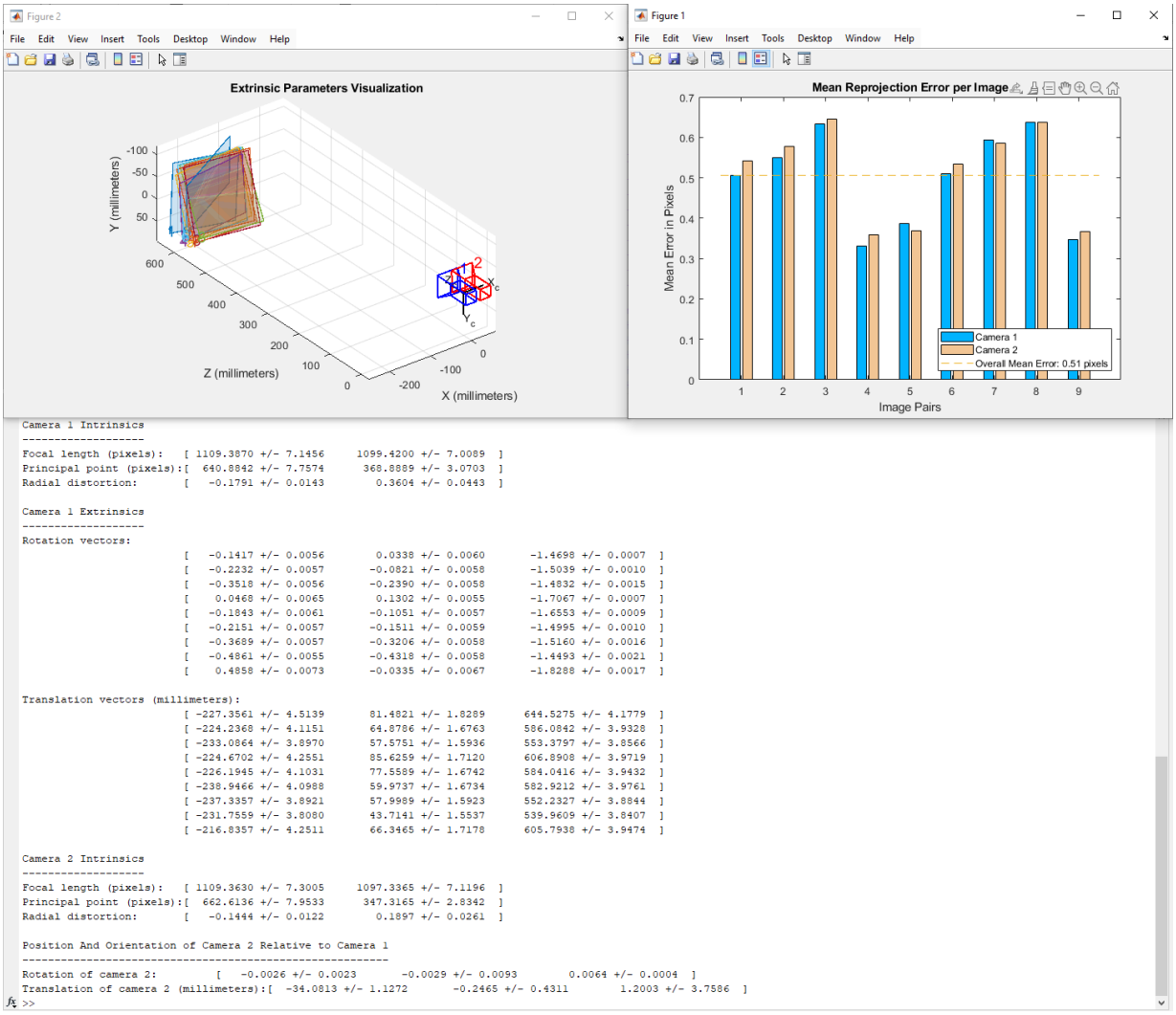

Figura 85. Visualización de los parámetros extrínsecos e intrínsecos

En Matlab la llamada dada por el *script* generado (Código 20), se encarga internamente de compensar la distorsión, aplicar la rotación de rectificación y re-muestrear los píxeles. La función devuelve las imágenes J1 y J2 listas para su visualización.

Código 20. Llamada a rectifyStereoImage()

```
% You can use the calibration data to rectify stereo images.
I2 = imread(imageFileNames2{1});
[J1, J2, reprojectionMatrix] = rectifyStereoImages(I1, I2, stereoParams);
```

Sin embargo, para implementar el algoritmo de rectificación en HLS, no se pueden pasar directamente las imágenes rectificadas como en Matlab, sino que el *hardware* deberá calcular, píxel a píxel, dónde leer cada componente de la imagen original para posteriormente construir

la imagen rectificada. Para ello, hacen falta dos matrices intrínsecas correspondientes a las cámaras originales; izquierda y derecha. También son necesarios los coeficientes de distorsión y las matrices de rectificación inversa que combinan la rotación de rectificación y la nueva matriz intrínseca de salida.

Esta lista de parámetros generados contiene sólo los correspondientes a la calibración, pero no las matrices rectificadas, por lo que, para la generación de matrices en formato punto flotante que el ejemplo HLS requiere, se hace uso de un *script* en Python (Código 21) que adapte la funcionalidad específica.

6.2.5. Generación de parámetros de rectificación

En la definición de los parámetros intrínsecos y de distorsión, la matriz intrínseca *K* 3x3 codifica la cámara *pin-hole*: focales f_x, f_y en píxeles y centro óptico c_x, c_y[47].

$$K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \tag{10}$$

El vector *D* contiene los coeficientes del modelo Brown-Conrady [48], usado para corregir la distorsión radial $[k_1 \pm {\sigma_k}^1, k_2 \pm {\sigma_k}^2]$ y tangencial $[p_1, p_2]$ de los objetivos. OpenCV utiliza el orden $D = [k_1, k_2, p_1, p_2, k_3]$. En este caso no se estiman p_1 ni p_2, por lo que a la hora de añadirlos al *script* quedan a cero.

Para los parámetros extrínsecos, Matlab genera un vector de Rodrigues, referente a la matriz de rotación *R,* que describe la orientación relativa de la cámara 2 respecto de la cámara 1. Con *cv2.Rodrigues(rvec2)* se obtiene la matriz 3x3 R tal que:

$$X_2 = RX_1;\ X_i = [x_i y_i z_i]^T \tag{11}$$

Por tanto, si un punto tiene coordenadas X_1en la cámara 1, el sistema de la cámara 2 es X_2.

OpenCV, a través del módulo *calib3d*, aporta las rutina de rectificación estereoscópica que la literatura considera estándar en vision por ordenador. Para la rectificación estéreo se hace

uso de la función *stereoRectify* inicial por la biblioteca *cv2*, que calcula las matrices de rotación para cada cámara, lo que prácticamente convierte ambos planos de imagen en uno mismo. En consecuencia, esto hace que todas las líneas epipolares sean paralelas, simplificando así el problema de correspondencia estéreo densa. La función toma como entrada las matrices calculadas a partir de la función *stereoCalibrate*. Como salida, proporciona dos matrices de rotación y dos matrices de proyección en las nuevas coordenadas [49]. La función distingue los casos de rectificación con líneas epipolares horizontales y verticales. En este caso para un par estéreo horizontal se tiene que la posición de la segunda cámara respecto a la primera es distinta en su mayoría en el eje X. Tras realizar la rectificación, las líneas epipolares de las imágenes izquierda y derecha quedarán en el plano horizontal, y se superpondrán en la misma coordenada Y. Así, las matrices de proyección P_1 y P_2, quedan con la siguiente estructura:

$$P_1 = \begin{bmatrix} f & 0 & cx_1 & 0 \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{12}$$

$$P_2 = \begin{bmatrix} f & 0 & cx_2 & T_x \cdot f \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{13}$$

$$Q = \begin{bmatrix} 1 & 0 & 0 & -cx_1 \\ 0 & 1 & 0 & -cy \\ 0 & 0 & 0 & f \\ 0 & 0 & -\frac{1}{T_x} & \frac{cx_1 - cx_2}{T_x} \end{bmatrix} \tag{14}$$

donde T_x es un desplazamiento horizontal entre las cámaras y $cx_1 = cx_2$ si se establece *CALIB_ZERO_DISPARITY*. Como se puede ver, las tres primeras columnas de P_1 y P_2 serán las nuevas matrices de cámara rectificadas. La función *remap* en HLS necesita como parámetro un bloque 3x3 de las dos matrices de proyección 3x4 y además su inversa $(A'R)^{-1}$, lo que, tras la rutina *stereoRectify*, convierten las coordenadas de píxel rectificado al espacio de cámara previo al muestreo de la imagen original.

Por último, se crea un *header* en el orden requerido por la IP de referencia (Código 22), de este modo, cada parámetro queda disponible como constante de síntesis que asegura la coherencia con los *kernels* de Vitis Vision (*xf::cv::Remap* y *StereoLBM*).

Código 21. Scrip python, generación de parámetros rectificado en formato punto flotante

```python
import numpy as np
import cv2

# Parámetros de la cámara desde MATLAB
# Cámara 1
fx1, fy1 = 800.7829, 800.8264
cx1, cy1 = 380.5913, 288.4567
k1_1, k2_1, p1_1, p2_1, k3_1 = 0.1448, -0.2084, 0.0, 0.0, 0.0

# Cámara 2
fx2, fy2 = 801.9180, 801.0330
cx2, cy2 = 376.8951, 272.8366
k1_2, k2_2, p1_2, p2_2, k3_2 = 0.2247, -0.5457, 0.0, 0.0, 0.0

# Rotación y traslación de cam2 respecto a cam1 (en mm)
rvec2 = np.array([-0.0042, 0.0189, 0.0065], dtype=np.float64)
tvec2 = np.array([-34.1991, -0.0271, 4.0830], dtype=np.float64).reshape(3, 1)

# Construcción de matrices
K1 = np.array([[fx1, 0, cx1],
               [0, fy1, cy1],
               [0, 0, 1]], dtype=np.float64)
K2 = np.array([[fx2, 0, cx2],
               [0, fy2, cy2],
               [0, 0, 1]], dtype=np.float64)
D1 = np.array([k1_1, k2_1, p1_1, p2_1, k3_1], dtype=np.float64)
D2 = np.array([k1_2, k2_2, p1_2, p2_2, k3_2], dtype=np.float64)

# Convertir el vector de rotación a matriz de rotación
# Usamos cv2.Rodrigues para convertir el vector de rotación a matriz
R, _ = cv2.Rodrigues(rvec2)

# Rectificación estéreo
image_size = (800, 600)
R1, R2, P1, P2, _, _, _ = cv2.stereoRectify(
    cameraMatrix1=K1, distCoeffs1=D1,
    cameraMatrix2=K2, distCoeffs2=D2,
    imageSize=image_size, R=R, T=tvec2,
    flags=cv2.CALIB_ZERO_DISPARITY, alpha=0
)

# Inversas de los sub-bloques 3×3 de P1 y P2
irA_l = np.linalg.inv(P1[:3, :3])
irA_r = np.linalg.inv(P2[:3, :3])
```

Continuación del script

```
# Formateo para C pnt flotante
def fmt(v):
    if abs(v) < 1e-11:
        return "0.00000000000"
    return f"{v:.11f}"

def c_array(name, arr, per_line):
    flat = arr.flatten()
    lines = []
    for i in range(0, len(flat), per_line):
        chunk = flat[i:i+per_line]
        lines.append("    " + ", ".join(fmt(val) for val in chunk))
    body = ",\n".join(lines)
    return f"const param_T {name}[{flat.size}] = {{\n{body}\n}};\n"

# Construcción del .h
guard = "_CAMERAPARAMETERS_"
h_lines = [
    f"#ifndef {guard}",
    f"#define {guard}\n",
    "typedef float param_T;\n",
    "// Matriz intrínseca izquierda",
    c_array("cameraMA_l", K1.astype(np.float32), 3),
    "// Matriz de rectificación inversa (R * A')-1 izquierda",
    c_array("irA_l", irA_l.astype(np.float32), 3),
    "// Coeficientes de distorsión izquierda",
    c_array("distC_l", D1.astype(np.float32), 5),
    "// Matriz intrínseca derecha",
    c_array("cameraMA_r", K2.astype(np.float32), 3),
    "// Matriz de rectificación inversa (R * A')-1 derecha",
    c_array("irA_r", irA_r.astype(np.float32), 3),
    "// Coeficientes de distorsión derecha",
    c_array("distC_r", D2.astype(np.float32), 5),
    f"#endif // {guard}\n"
]

# Con encoding UTF-8
with open("cameraParameters.h", "w", encoding="utf-8") as f:
    f.write("\n".join(h_lines))

print("Archivo generado: cameraParameters.h")
```

Código 22. Parámetros de rectificación generados

```
typedef float param_T;

// Matriz intrínseca izquierda
const param_T cameraMA_l[9] = {
    800.78289794922, 0.00000000000, 380.59130859375,
    0.00000000000, 800.82641601562, 288.45669555664,
    0.00000000000, 0.00000000000, 1.00000000000
};

// Matriz de rectificación inversa (R * A')-1 izquierda
const param_T irA_l[9] = {
    0.00114964868, 0.00000000000, -0.55973756313,
    0.00000000000, 0.00114964868, -0.32182618976,
    0.00000000000, 0.00000000000, 1.00000000000
};

// Coeficientes de distorsión izquierda
const param_T distC_l[5] = {
    0.14480000734, -0.20839999616, 0.00000000000, 0.00000000000, 0.00000000000
};

// Matriz intrínseca derecha
const param_T cameraMA_r[9] = {
    801.91802978516, 0.00000000000, 376.89511108398,
    0.00000000000, 801.03302001953, 272.83660888672,
    0.00000000000, 0.00000000000, 1.00000000000
};

// Matriz de rectificación inversa (R * A')-1 derecha
const param_T irA_r[9] = {
    0.00114964868, 0.00000000000, -0.55973756313,
    0.00000000000, 0.00114964868, -0.32182618976,
    0.00000000000, 0.00000000000, 1.00000000000
};

// Coeficientes de distorsión derecha
const param_T distC_r[5] = {
    0.22470000386, -0.54570001364, 0.00000000000, 0.00000000000, 0.00000000000
};
```

6.3. RECTIFICACIÓN EN TIEMPO REAL SOBRE EL IP *STEREOLBM-AXIS*

6.3.1. Implementación hardware: Función *remap()*

Se ha comprobado que, por cuestiones de limitación de recursos de la placa, la función *remap* no se puede utilizar con la resolución de 1280 × 720 considerada inicialmente. En la implementación del diseño Vivado (*place design*), se requerían más celdas LUT de RAM distribuida que las disponibles en el dispositivo. En concreto, este diseño requeriría ≈40k de estos tipos de celdas, que excedían los disponibles en el dispositivo.

Por tanto, se determinó que la opción más viable inicialmente era reducir el modo de vídeo a 800 x 600 píxeles. Para ello se vuelve a realizar el proceso de calibración y rectificación *offline,* donde se comprueba que los parámetros intrínsecos y extrínsecos se ajustan adecuadamente a la nueva resolución (Figura 86), buscando la máxima precisión.

```
Camera 1 Intrinsics
-------------------
Focal length (pixels):   [  800.7829 +/- 12.5842     800.8264 +/- 13.2248 ]
Principal point (pixels):[  380.5913 +/- 5.3932      288.4567 +/- 1.8249  ]
Radial distortion:       [    0.1448 +/- 0.0200       -0.2084 +/- 0.0600  ]

Camera 1 Extrinsics
-------------------
Rotation vectors:
                         [    0.1117 +/- 0.0056       -0.0028 +/- 0.0038       -1.5520 +/- 0.0005  ]
                         [    0.1806 +/- 0.0069       -0.1366 +/- 0.0046       -1.4478 +/- 0.0007  ]
                         [    0.1964 +/- 0.0072       -0.0857 +/- 0.0044       -1.3682 +/- 0.0009  ]
                         [   -0.1175 +/- 0.0048       -0.2804 +/- 0.0044       -1.5202 +/- 0.0008  ]
                         [   -0.1281 +/- 0.0048       -0.3770 +/- 0.0047       -1.5008 +/- 0.0010  ]
                         [    0.0370 +/- 0.0053       -0.2448 +/- 0.0051       -1.4331 +/- 0.0005  ]

Translation vectors (millimeters):
                         [ -190.3776 +/- 3.9375       20.1813 +/- 1.3007      576.4859 +/- 9.5363  ]
                         [ -188.6954 +/- 4.0338       -8.0661 +/- 1.3287      582.8466 +/- 9.6980  ]
                         [ -217.3597 +/- 4.0188      -10.5511 +/- 1.3158      581.2404 +/- 9.6967  ]
                         [ -199.7732 +/- 3.9756       -6.6728 +/- 1.3059      575.0360 +/- 9.6503  ]
                         [ -207.4454 +/- 3.8725      -26.3560 +/- 1.2616      557.4690 +/- 9.4234  ]
                         [ -203.7592 +/- 4.4502       -6.6439 +/- 1.4693      645.1186 +/- 10.7314 ]

Camera 2 Intrinsics
-------------------
Focal length (pixels):   [  801.9180 +/- 12.5801     801.0330 +/- 13.1682 ]
Principal point (pixels):[  376.8951 +/- 5.1773      272.8366 +/- 1.7801  ]
Radial distortion:       [    0.2247 +/- 0.0216       -0.5457 +/- 0.0690  ]

Position And Orientation of Camera 2 Relative to Camera 1
---------------------------------------------------------
Rotation of camera 2:          [   -0.0042 +/- 0.0027        0.0189 +/- 0.0060        0.0065 +/- 0.0004  ]
Translation of camera 2 (millimeters):[  -34.1991 +/- 0.7317       -0.0271 +/- 0.5060        4.0830 +/- 3.2617  ]
```

Figura 86. Parámetros finales de calibración para la resolución 800x600

Una vez recalibrados los parámetros, se pasa al flujo de trabajo HLS. Se ha optado por adaptar el ejemplo *stereopipeline* incluido en librería de Vitis Visión, incorporando dentro del *kernel* HLS las funciones *xf::cv::InitUndistortRectifyMapInverse()* y *xf::cv::remap()* entregadas por la librería.

El flujo de procesamiento que se ha implementado se muestra en la Figura 87, de tal forma que cada imagen de entrada (izquierda y derecha) se somete a un bloque de rectificación (que se sitúa dentro del recuadro punteado), el cual está compuesto por la inicialización del mapa de remapeo inverso seguida de la operación de remapeo, posterior a la cual ambas imágenes rectificadas se proporcionan al módulo cálculo de disparidad (StereoBM) [50].

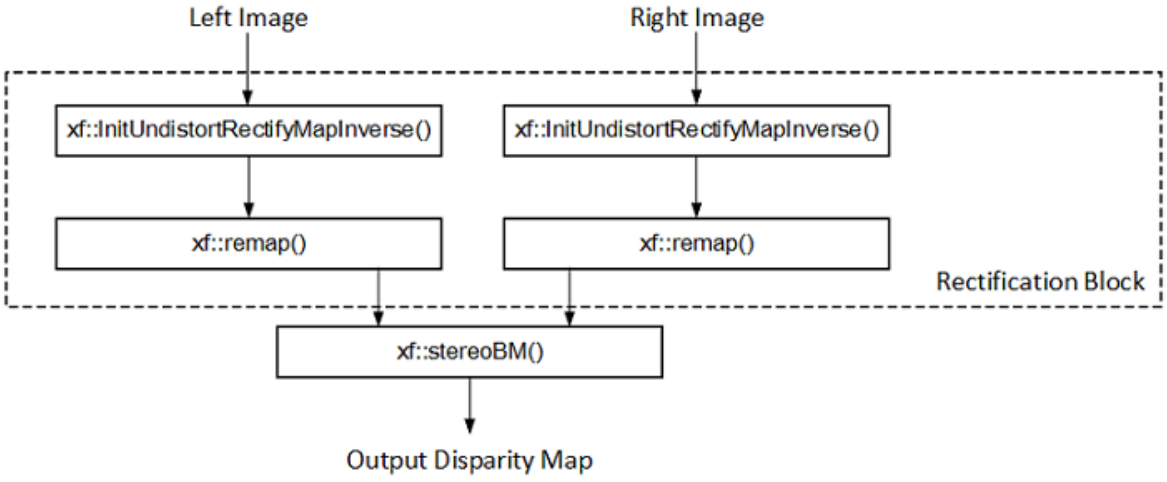

Figura 87. Diagrama de bloques del proceso de rectificación estereoscópica y cálculo de disparidad usando funciones hardware de Vitis Vision.

En el código *xf_stereolbm_accel.cpp*, la rectificación geométrica se implementa en *hardware* dentro del *kernel* HLS *stereolbm_axis_cambm*. En primer lugar, se parametrizan los valores de `preFilterCap`, `uniquenessRatio` y `textureThreshold` (Código 23) mediante la interfaz de control *AXI-Lite*, posibilitando que la aplicación *bare-metal* pueda modificar estos valores en tiempo de ejecución, y ajustar así la respuesta de la técnica de *block matching*. Esto permite poder experimentar con combinaciones de pre-filtrado, ratios para eliminar coincidencias ambiguas y umbrales de textura con el fin de eliminar zonas de baja información. Todo sin necesidad de volver a sintetizar el diseño *hardware*.

Código 23. Directivas HLS de preFilterCap, uniquenessRatio y textureThreshold

```
#pragma HLS INTERFACE s_axilite port=preFilterCap bundle=Ctrl
#pragma HLS INTERFACE s_axilite port=uniquenessRatio bundle=Ctrl
#pragma HLS INTERFACE s_axilite port=textureThreshold bundle=Ctrl
```

A continuación, se cargan los parámetros de calibración pre-calculados (en el apartado 6.2.5. Generación de parámetros de rectificación) definidos en el fichero *cameraParameters.h*. Estos valores como, por ejemplo: *cameraMA_l, distC_l, irA_l,* para la cámara izquierda, son copiados a variables internas de tipo aritmético fijo *ap_fixed<32,12>* [51] (Código 24) para ajustarlos al *hardware*. Posteriormente, para realizar las transformaciones geométricas aplicadas por cada cámara, se llama a *xf::cv::InitUndistortRectifyMapInverse* pasando las matrices de calibración intrínseca, de distorsión y de rectificación que calculan los mapas de remapeo (*mapx* y *mapy*) correspondientes a dicha transformación geométrica (Código 24). La función *hardware* genera dos mapas en punto fijo (objetos *xf::cv::Mat* de tipo XF_32FC1) que, para cada coordenada del plano rectificado, indican su posición correspondiente en la imagen original distorsionada. Dichos mapas actúan como *look-up tables* de corrección geométrica.

Código 24. Declaración de matrices de calibración en ap fixed

```
// clang-format on
ap_fixed<32, 12>
cameraMA_l_fix[XF_CAMERA_MATRIX_SIZE],cameraMA_r_fix[XF_CAMERA_MATRIX_SIZE],
distC_l_fix[XF_DIST_COEFF_SIZE], distC_r_fix[XF_DIST_COEFF_SIZE],
irA_l_fix[XF_CAMERA_MATRIX_SIZE], irA_r_fix[XF_CAMERA_MATRIX_SIZE];
```

Tras calcular los mapas, se pasa a utilizar la función *xf::cv::remap* para re-mapear la imagen de entrada original en la nueva imagen rectificada. Este módulo toma cada píxel de la imagen de entrada (*imgL_in* e *imgR_in*) y lo vuelve a colocar en la imagen de salida en la posición correspondiente a la imagen rectificada (*leftRemappedMat* y *rightRemappedMat*), de acuerdo con las coordenadas del mapa (*mapx* y *mapy*). En caso de que las coordenadas de remapeo no coincidan exactamente con el centro de un píxel, la intensidad de salida se calcula mediante interpolación a partir de los píxeles vecinos, sin necesidad de redondear las coordenadas. De este modo, ambas operaciones (*InitUndistortRectifyMapInverse* + *remap*) están configuradas para trabajar con imágenes de un solo canal (formato 8UC1 para escala de grises) y aprovechando el paralelismo de nivel píxel (NPPC=1).

Antes de pasar a su síntesis como un núcleo IP, se opta por incluir en la implementación *hardware* los procesos morfológicos de erosión y dilatación, con el fin de poder analizar su influencia para mejorar la calidad de la imagen final.

6.3.2. Implementación hardware: Procesos morfológicos de erosión y dilatación.

Las operaciones morfológicas configuran una amplia familia de procedimientos para procesar imágenes en función de sus formas geométricas, las cuales se obtienen mediante la aplicación de un elemento estructurante a la imagen original con el objetivo de obtener una nueva imagen del mismo tamaño. Las más elementales, y las que se utilizarán en la implementación *hardware* son: la dilatación y la erosión. La dilatación es capaz de añadir píxeles al contorno de los objetos, y la erosión es capaz de quitarlos. La cantidad de píxeles añadiéndose o quitándose dependerá de la forma y del tamaño del elemento estructurante a utilizar, quedando en el caso de ambos procedimientos el valor de cada píxel de la imagen resultante. El estado de cualquier píxel de la imagen de salida se determina aplicando una regla al píxel correspondiente, y a sus vecinos de la imagen de entrada [52].

En el acelerador HW implementado, una vez obtenida la imagen de disparidad rectificada, se añaden operaciones morfológicas al flujo *hardware* para intentar mejorar la calidad y robustez del mapa de profundidad obtenido. Se aplica la operación de apertura morfológica (erosión seguida de dilatación) sobre la imagen de disparidad en escala de grises. Esta operación trata de reducir el ruido aislado, rellenar pequeños huecos del mapa de disparidad y suavizar los contornos de los objetos. Para implementarla en la sección PL del dispositivo ZYNQ se utilizaron los módulos *xf::cv::erode* y *xf::cv::dilate* (Código 25).

Código 25. Implementación HLS de la apertura morfológica y conversión de resultado final

```
    // Erode:
  xf::cv::Mat<IN_TYPE, MAX_HEIGHT, MAX_WIDTH, NPPCX, XF_CV_DEPTH_OUT>
img_disp8u_erode(rows, cols);
    xf::cv::erode<XF_BORDER_CONSTANT, IN_TYPE, MAX_HEIGHT, MAX_WIDTH, KERNEL_SHAPE,
FILTER_SIZE, FILTER_SIZE, ITERATIONS, NPPCX, XF_CV_DEPTH_OUT, XF_CV_DEPTH_OUT>(img_disp8u,
img_disp8u_erode, _kernel);

    // Dilate:
  xf::cv::Mat<IN_TYPE, MAX_HEIGHT, MAX_WIDTH, NPPCX, XF_CV_DEPTH_OUT>
img_disp8u_dilate(rows, cols);
    xf::cv::dilate<XF_BORDER_CONSTANT, IN_TYPE, MAX_HEIGHT, MAX_WIDTH, KERNEL_SHAPE,
FILTER_SIZE, FILTER_SIZE, ITERATIONS, NPPCX, XF_CV_DEPTH_OUT, XF_CV_DEPTH_OUT>
(img_disp8u_erode, img_disp8u_dilate, locKernel);

    // Convert _dst xf::Mat object to output array:
    xf::cv::xfMat2AXIvideo(img_disp8u_dilate, vid_out);
```

Por tanto, tras la optimización del acelerador que incluye la incorporación de los nuevos módulos de apertura morfológica a la salida del núcleo *StereoBM*, el *pipeline hardware* resultante se resume así (Figura 88):

El sistema recibe en paralelo dos flujos *AXI4-Stream*, uno de la cámara izquierda y otro de la cámara derecha, y cada flujo es transformado en *xf::Mat* y, simultáneamente, rectificado siguiendo los parámetros de calibración para corregir distorsiones y alinear las imágenes tomadas. Las imágenes rectificadas se proporcionan al motor *Stereo Local Block Matching* (SLBM) para obtener un mapa de disparidad de 16 bits donde cada píxel tiene un valor inversamente proporcional a la distancia. A este mapa crudo de disparidad se le aplica un procesado posterior, comenzando por un escaldado y su conversión a 8 bits mediante *ConvertShiftAbs* para después ejecutar una secuencia de una apertura morfológica formada por los pasos de erosión y dilatación, eliminando así el ruido y rellenando los huecos. Por último, se vuelve a empaquetar en un flujo *AXI4-Stream* para su transmisión hacia el resto del sistema. El producto resultante representa el mapa de profundidad.

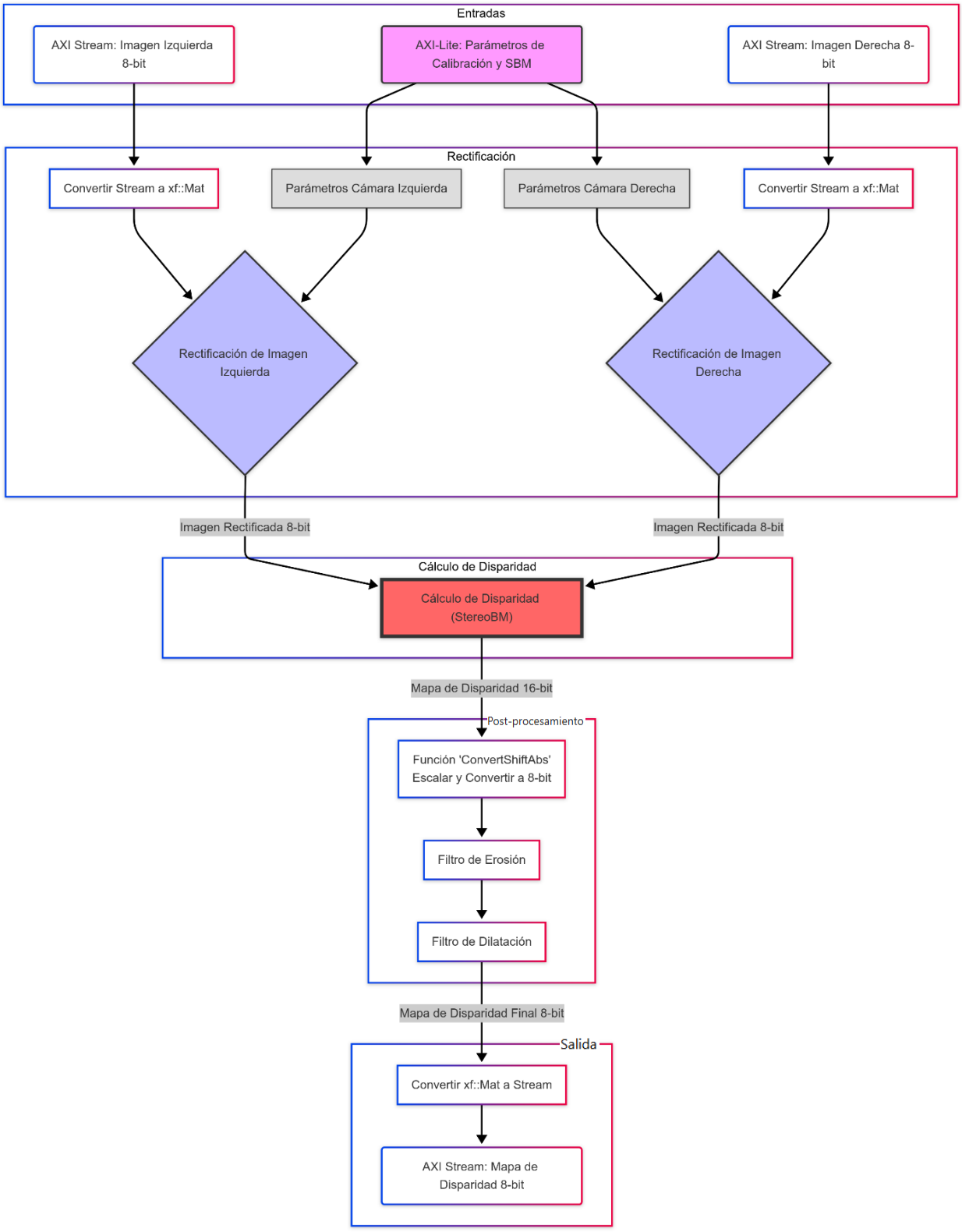

Figura 88. Pipeline de Procesamiento del Acelerador Hardware de Visión Estéreo

6.3.3. Generación del IP *StereoLBM* final de la plataforma

Para generar el núcleo del IP HLS SLBM definitivo, se sigue el mismo proceso descrito en el apartado 5.3. Flujo de generación y verificación del IP, haciendo uso de un *script* tcl que automatiza el proceso. Así, tras finalizar la síntesis y la posterior implementación *Place & Route*, la ocupación de recursos queda resumida en la Tabla.

Tabla 6. Resumen de implementación de recursos

Design Summary
impl_1
xc7z020clg400-1

Criteria	Guideline	Actual	Status
LUT	70%	52.48%	OK
FD	50%	26.40%	OK
LUTRAM+SRL	25%	10.16%	OK
MUXF7	15%	0.48%	OK
LUT Combining	20%	13.62%	OK
DSP	80%	90.91%	REVIEW
RAMB/FIFO	80%	66.43%	OK
DSP+RAMB+URAM (Avg)	70%	78.67%	REVIEW
BUFGCE* + BUFGCTRL	24	0	OK
DONT_TOUCH (cells/nets)	0	0	OK
MARK_DEBUG (nets)	0	0	OK
Control Sets	998	391	OK
Average Fanout for modules > 100k cells	4	0	OK
Non-FD high fanout nets > 10k loads	0	0	OK

Una vez completada la etapa de *Place & Route*, el IP *StereoLBM* utiliza los siguientes recursos lógicos:

- Se reservan 10134 SLICEs, de los cuales una buena parte se utilizan para contener 27917 LUTs en la lógica combinacional.
- Se utilizan 28090 Flip-Flops para almacenar resultados intermedios.
- Para poder acelerar los cálculos de la correlación estereoscópica, se utilizan 200 bloques DSP dedicados, llegando casi al límite de la capacidad de la placa.
- La memoria en chip es soportada por 186 BRAMs de 36 Kb que se utilizan como *buffers* de píxeles y ventanas de procesamiento.
- Se usan 1724 SRLs que indican pequeñas memorias intermedias implementadas dentro de LUTs para optimizar la lógica de *pipeline*.

El diseño ocupa preferentemente LUTs y FFs dentro de los SLICE, el uso intensivo de DSPs para acelerar los cómputos de la correlación estereoscópica, y de BRAM para los *buffers* (Tabla 7).

Tabla 7. Uso de recursos post implementación

```
#=== Post-Implementation Resource usage ===
SLICE:        10134
LUT:          27917
FF:           28090
DSP:            200
BRAM:           186
URAM:             0
LATCH:            0
SRL:           1724
CLB:              0

#=== Final timing ===
CP required:                    10.000
CP achieved post-synthesis:     7.737
CP achieved post-implementation: 9.622
Timing met
```

6.3.4. Cadena completa y aplicación *software* final

Siguiendo el proceso realizado en el apartado de 5.4.1. Incorporación en el diagrama de bloques del Capítulo 5, se genera la síntesis e implementación del diseño final en el entorno Vivado. La Figura 89 muestra el diagrama de bloques del diseño de la plataforma y la Figura 90 muestra los recursos utilizados tras la implementación del diseño en el dispositivo.

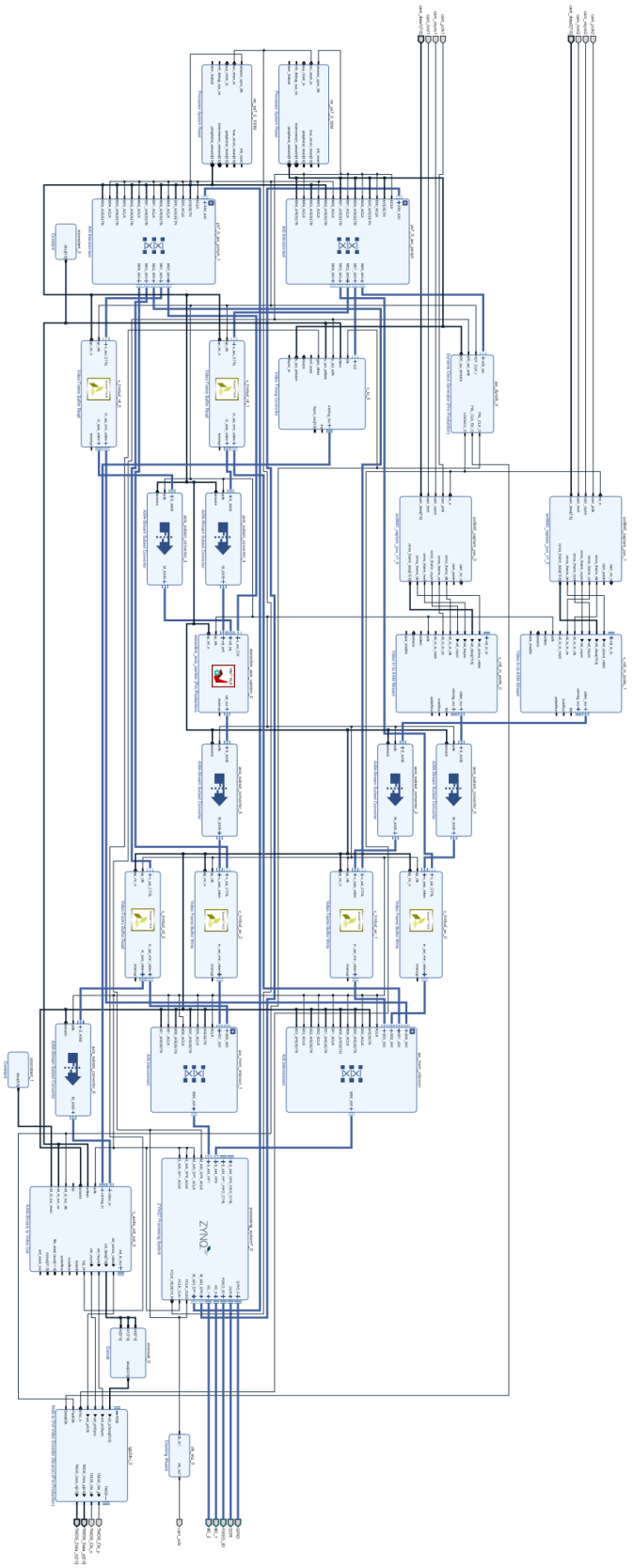

Figura 89. Diseño final del diagrama de bloques de la arquitectura del sistema

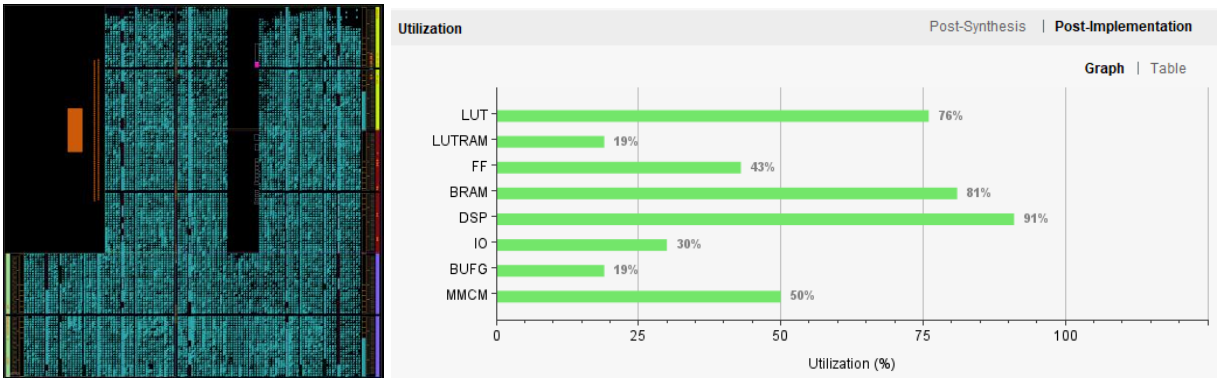

Figura 90. Diseño implementado y utilización de recursos final

Por su parte, en la aplicación SW, tras exportar el *hardware* final de la plataforma, se instancia el nuevo módulo de abstracción en C que encapsula la configuración y la inicialización del núcleo *hardware* (*xstereolbm_axis_cambm.c*), adaptado en el fichero *sv_ctrl.c*. El IP *stereolbm_axis_cambm*, es inicializado en la función *StereoVisionCtrl_init()* y los registros `textureThreshold`, `uniquenessRatio` y `preFilterCap` se parametrizan dentro de la función *ApplyStereoParams()*.

Para una respuesta de vídeo más fluida, una mejora realizada con respecto a la plataforma previa fue el cambio de doble a triple *buffer* dentro de la lógica de los bloques *Video Frame Buffers*. Cada canal mantiene tres búferes en la memoria: uno para lectura, otro para escritura y un tercero "libre". Las funciones principales son:

- Inicialización (*FrameBufferStartTripleX*): Se establecen los índices iniciales: `readIdx = 0` (*buffer* que será el primero en mostrarse), `writeIdx = 1` (*buffer* donde se volcará la primera trama), `freeIdx = 2` (el *buffer* libre). Luego, se configuran respectivamente la IP de lectura (*XV_frmbufrd*) apuntando hacia *bufAddr[readIdx]* y la IP de escritura (*XV_frmbufwr*) apuntando hacia *bufAddr[writeIdx]* mediante llamadas a *FrameBufferReadSetupX* y *FrameBufferWriteSetupX*.
- Rotación de *buffers* (*FrameBufferCheckWriteTripleX*): Se comprueba si la IP de escritura ha dejado de estar activa (*IsIdle*). En el caso de que haya un búfer completo se

proceden a rotar los índices de forma cíclica. Finalmente, se actualiza la IP de lectura a *bufAddr[readIdx]* y se relanza la de escritura en *bufAddr[writeIdx]*.

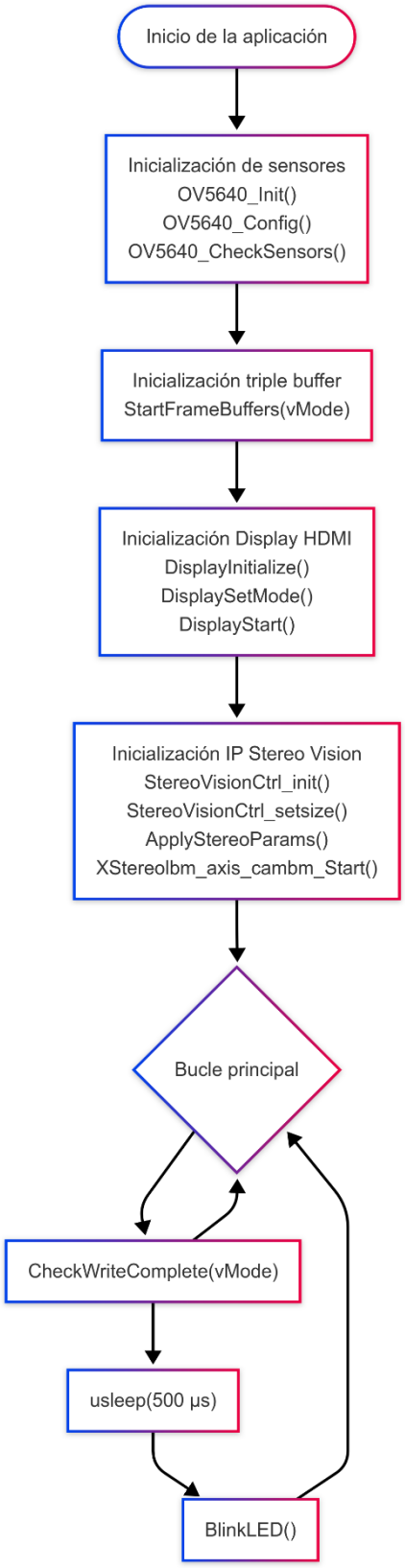

Figura 91. diagrama de flujo de la cadena de vídeo final

La Figura 91 muestra el diagrama de flujo de la cadena de vídeo final, tal como se implementa en la función *main()* del archivo *main.c*:

1. Sensores OV5640. Ambos sensores se inicializan y se configuran a través de su interfaz SCCB: Las rutinas mencionadas hacen arrancar el bus de I²C/SCCB, cargan los registros de configuración y chequean el ID de los sensores antes de poder continuar.

```
// Seleccionamos la resolución de vídeo
vMode = VMODE_800x600;

// OV5640, inicio, configuración y verificiación del ID del sensor
OV5640_Init();
OV5640_Config();
if (OV5640_CheckSensors() != XST_SUCCESS) {
    return -1;
}
```

2. Triple *buffering*. Con *StartFrameBuffers(vMode)* se reservan tres *buffers* por flujo (izquierda, derecha y disparidad) y se inicia la ejecución de los *Frame Buffers* en modo *triple buffer*, asegurando siempre un *buffer* de lectura, uno de escritura y otro libre para evitar el *tearing*.
3. Display HDMI. El controlador de vídeo se parametriza y se inicia su ejecución: Ajusta el *pixel clock*, los *timings* correspondientes a una resolución de 800×600@60 Hz y habilita la salida por HDMI/DVI.

```
// Display, inicializar y arrancar
DisplayInitialize(&dispCtrl, DISP_VTC_ID, DYNCLK_BASEADDR);
DisplaySetMode(&dispCtrl, &vMode);
DisplayStart(&dispCtrl);
```

4. IP de *Stereo Vision*. El bloque *hardware* de *stereo matching* se configura y se inicia su ejecución: Se inicializa el *pipeline* del *block matching* en paralelo con la adquisición de nuevas tramas.

```
// Stereovision, inicializar, ajustar tamaño, parámetros y arrancar
StereoVisionCtrl_init(&sv, STEREOVISION_ID);
StereoVisionCtrl_setsize(&sv, vMode.width, vMode.height);
ApplyStereoParams(&sv, &params);
XStereolbm_axis_cambm_Start(&sv);
```

5. Bucle principal: a partir de aquí se monitoriza la escritura de los *buffers* (*CheckWriteComplete*) haciendo el *swap* en caso necesario y, manteniendo así un flujo continuo.

Código 26. Bucle prinicpal de la función main()

```
while(1)
{
    //IP Write check fin de frame y swap
    CheckWriteComplete(vMode);
    // Espera (0,5 ms)
    usleep(500);
    // Parpadeo de LED cada ~500 ms
    BlinkLED();
}
return 0;
```

Como se puede observar en la Figura 92, el mapa de disparidad ha mejorado pasando de una distribución inicial con ruido disperso a una con contornos más nítidos. Las regiones de interés se muestran con mayor continuidad y la densidad de puntos erróneos disminuye sustancialmente, especialmente en la región media y sobre las orillas verticales. Esto refleja la mejora tras la calibración y la rectificación que se han aplicado. Cabe mencionar que, pese a que se aplicaron procesos de dilatación y erosión, no se han comprobado diferencias significativas en el resultado observado. La captura corresponde a un plano cenital del escritorio y muestra con claridad los contornos de las teclas del teclado, la superficie de la mesa, la pantalla y otros objetos presentes.

Por su parte, en las Figura 93 a 96 se ilustra la evolución del mapa de disparidad al situar el damero a cuatro distancias cada vez mayores con respecto al plano de las cámaras. Puede apreciarse que el patrón pasa de tonos esencialmente blancos a niveles de gris cada vez más oscuros conforme aumenta la distancia, lo que evidencia la disminución progresiva de la disparidad estimada. Esto inidica que el sistema responde de manera coherente a variaciones métricas y que la calibración efectuada conserva la proporcionalidad esperada entre disparidad y profundidad. En su conjunto, los resultados confirman que la plataforma puede distinguir escalones de distancia dentro del rango analizado con un error residual reducido y contornos estabilizados entre todas las capturas.

Figura 92. Captura del mapa de disparidad final – plano cenital

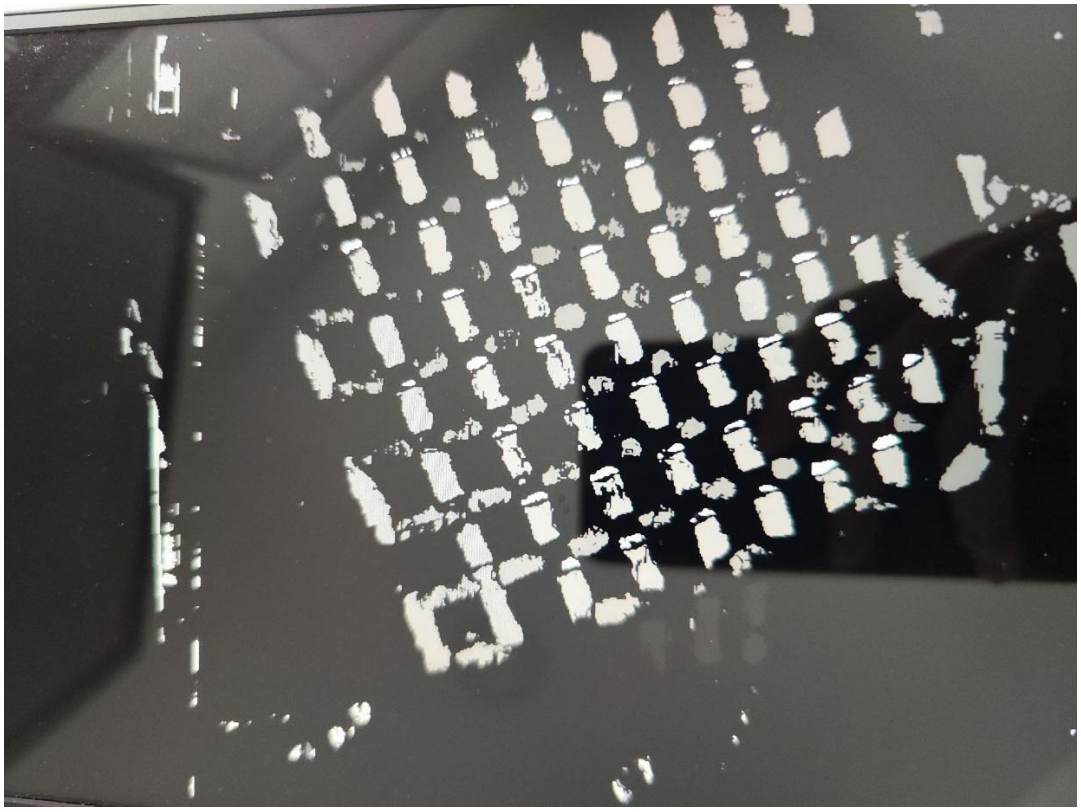

Figura 93. Mapa de disparidad — damero distancia 1

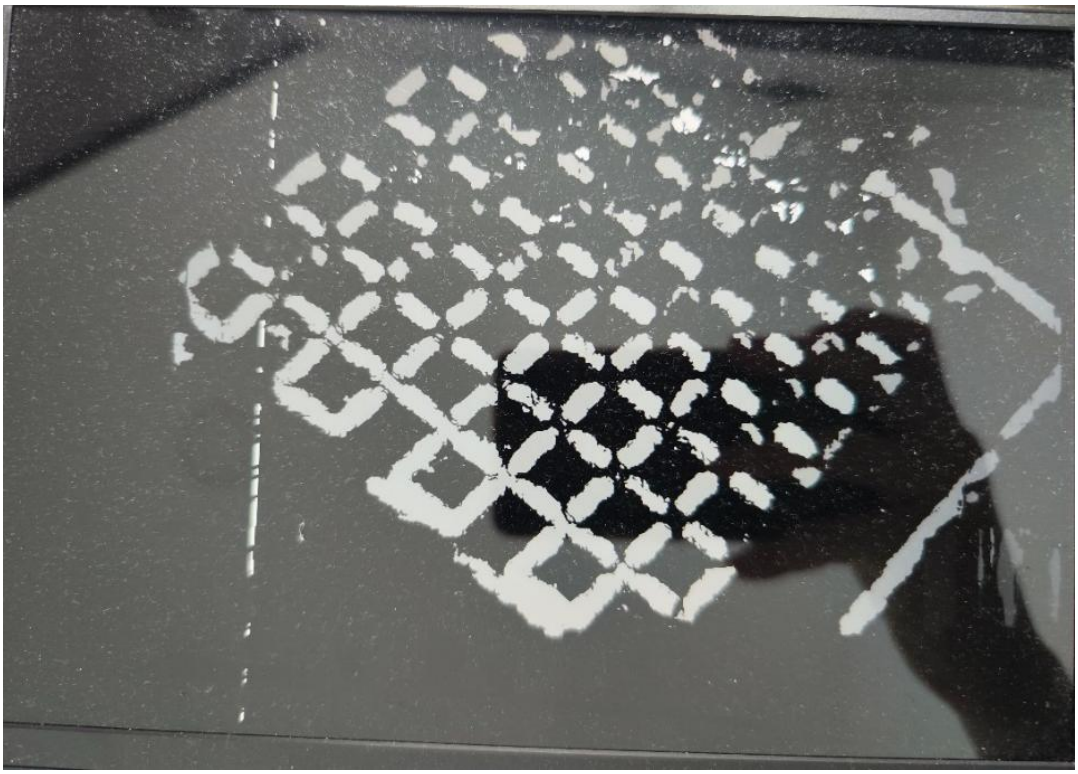

Figura 94. Mapa de disparidad — damero distancia 2

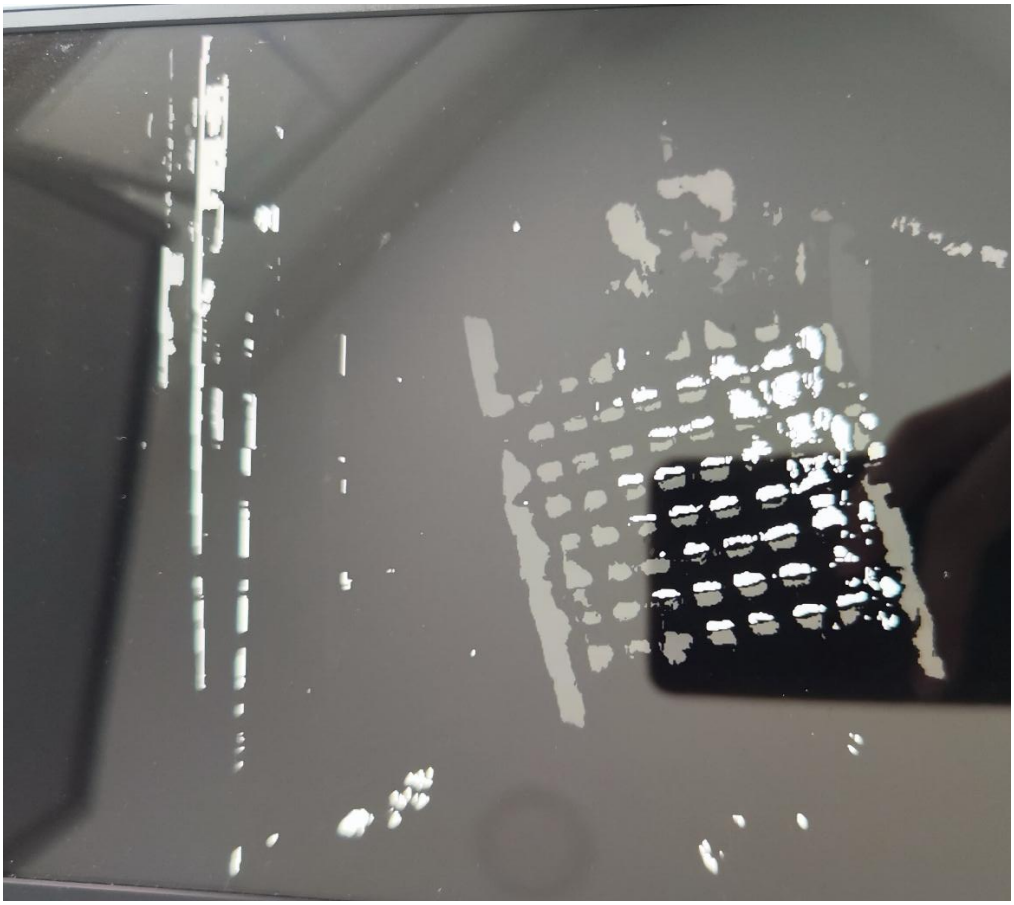

Figura 95. Mapa de disparidad — damero distancia 3

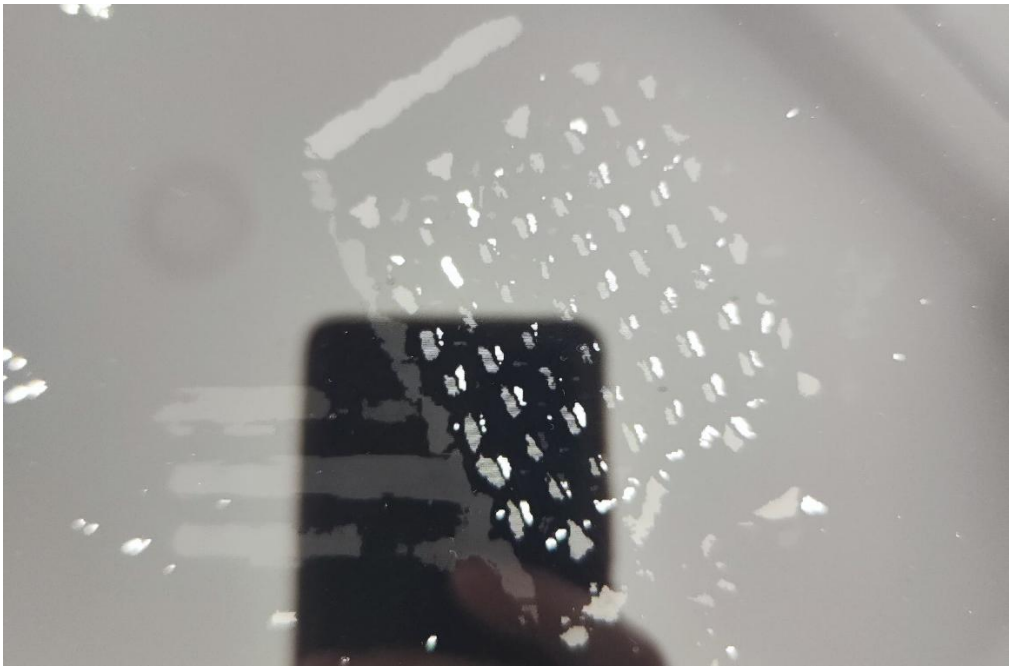

Figura 96. Mapa de disparidad — damero distancia 4

CAPÍTULO 7. CONCLUSIONES

En este capítulo se exponen las conclusiones y trabajos futuros del presente Trabajo de Fin de Grado.

En el desarrollo de este TFG se ha implementado una plataforma de percepción estereoscópica de profundidad basada en un SoPC que ha permitido la implementación de un sistema funcional para obtener información tridimensional de escenas reales. Para ello, se ha utilizado un dispositivo Zynq-7000 que integra el procesador ARM *Dual Core* con lógica programable FPGA, lo que permite adquirir en paralelo y procesar en *hardware* los datos de dos cámaras CMOS OV5640.

El sistema implementado en la placa de prototipado PYNQ-Z2 que se configura a través del protocolo SCCB, la calibración geométrica *offline* de las cámaras y la generación del mapa de disparidad mediante un algoritmo de *block matching* local (SLBM). Todo ello implementado en bloques IP *ad-hoc* y proporcionados en el entorno Vivado 2023.1, sintetizados sobre FPGA. Asimismo, se ha diseñado una PCB dedicada para el conjunto de cámaras, que optimiza la integridad de la señal y que facilita la integración física con la placa de prototipado PYNQ-Z2. El proceso de desarrollo ha integrado herramientas de diseño *hardware* y *software* (Vivado, Vitis HLS y OpenCV), permitiendo sintetizar una arquitectura heterogénea en la cual la lógica programable acelera el procesamiento estéreo y el procesador ejecuta las tareas de control, configuración y visualización. Se ha conseguido así un prototipo capaz de capturar imágenes estéreo en tiempo real mientras genera los correspondientes mapas de disparidad de forma consistente. En la evaluación realizada con patrones calibrados, se ha comprobado que la intensidad de la disparidad disminuye progresivamente con la distancia, mostrando que el sistema es capaz de distinguir correctamente los diferentes planos de profundidad con un error

residual bajo. Todos estos logros argumentan que la plataforma implementada cumple con los objetivos que se plantearon y, así, prueban la viabilidad técnica de la propuesta defendida.

En cuanto a las limitaciones del sistema actual, se ha observado que la implementación disponible requiere prácticamente de la totalidad de algunos de los tipos de los recursos de la FPGA, empleando cerca del límite las unidades DSP y LUT del dispositivo ZYNQ integrado en la placa de prototipado PYNQ-Z2. Esto ha imposibilitado la ejecución completa de ciertas etapas de rectificación en *hardware*; por ejemplo, no ha sido factible aplicar la función de remapeo de imagen a resolución completa (1280×720) debido a la escasez de celdas de memoria distribuidas. Por otro lado, el uso de un algoritmo *local de block matching*, aunque también en términos de lógica es más eficiente, tiene limitaciones en escenas con texturas homogéneas o una baja iluminación. Esto perjudica la calidad de imagen, lo que puede traducirse en artefactos o menor precisión en la estimación de disparidad.

Se plantean como mejoras y líneas futuras de trabajo fomentar el crecimiento del sistema a partir de algoritmos de correspondencia estéreo más avanzados (*Semi-Global Matching* o modelos basados en *Deep Learning*) que permitan obtener un mapa de profundidad superior. Por otro lado, en relación con las limitaciones *hardware* actuales, se propone mejorar el diseño de bloques críticos (fragmentando las rutas de mayor longitud o utilizando módulos más eficientes) que permitan aumentar la frecuencia máxima de operación y reducir el uso de recursos DSP y LUT. Otra línea de mejora que se puede contemplar es la migración de la solución a plataformas de mayores prestaciones (p.ej. MPSoC UltraScale+) que permitan utilizar resoluciones más altas, más cámaras, o algoritmos más complejos sin restricciones.

Por último, se plantea la posibilidad de investigar la integración de la plataforma en aplicaciones concretas, como la inserción en sistemas robóticos o en vehículos autónomos, incrementando la calidad de la interfaz de usuario y estudiando el consumo energético en entornos dinámicos de la vida real.

En el transcurso de este Trabajo Fin de Grado se ha podido validar la viabilidad de la plataforma diseñada, habiéndose cumplido así los objetivos inicialmente establecidos en cuanto a integración *hardware-software* y generación de información tridimensional. A lo largo de este TFG se han afianzado aprendizajes destacados en el diseño de sistemas embebidos y visión artificial, dejando también distinguir la relevancia de un enfoque metodológico incremental. De

la propia elaboración de los resultados, se han podido extraer claramente oportunidades de mejora, que permitirán la evolución de la plataforma desarrollada.

BIBLIOGRAFÍA

[1] S. Martín, J. Suárez, … R. R.-O. U., y undefined 2013, «Aplicación de los sistemas de visión estereoscópica en las enseñanzas técnicas», *researchgate.net*, Accedido: 18 de enero de 2025. [En línea]. Disponible en: https://www.researchgate.net/profile/Ramon-Rubio/publication/228779560_Aplicacion_de_los_sistemas_de_vision_estereoscopica_en_las_ensenanzas_tecnicas/links/00b7d52be0af39d406000000/Aplicacion-de-los-sistemas-de-vision-estereoscopica-en-las-ensenanzas-tecnicas.pdf

[2] «Configuring Stereo Depth». Accedido: 27 de enero de 2025. [En línea]. Disponible en: https://docs.luxonis.com/hardware/platform/depth/configuring-stereo-depth/

[3] Xilinx y Inc, «Zynq-7000 All Programmable SoC: Concepts, Tools, and Techniques (CTT) A Hands-On Guide to Effective Embedded System Design», 2013, Accedido: 31 de enero de 2025. [En línea]. Disponible en: http://www.xilinx.com/warranty.htm#critapps.

[4] M. Montalvo Martínez, «Técnicas de visión estereoscópica para determinar la estructura tridimensional de la escena», 2010. Accedido: 18 de enero de 2025. [En línea]. Disponible en: https://hdl.handle.net/20.500.14352/46251

[5] S. Kumar, C. Micheloni, C. Piciarelli, y G. L. Foresti, «Stereo rectification of uncalibrated and heterogeneous images», *Pattern Recognit Lett*, vol. 31, n.º 11, pp. 1445-1452, ago. 2010, doi: 10.1016/J.PATREC.2010.03.019.

[6] «OpenCV: Camera Calibration». Accedido: 18 de enero de 2025. [En línea]. Disponible en: https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html

[7] L. Ziwei, C. Wu, D. Chen, L. Shasha, y X. Yu, «An improved block matching motion estimation algorithm based on adaptive rood pattern search», *Proceedings of the 29th Chinese Control and Decision Conference, CCDC 2017*, pp. 5199-5203, jul. 2017, doi: 10.1109/CCDC.2017.7979419.

[8] «Vitis Vision Library». Accedido: 8 de julio de 2025. [En línea]. Disponible en: https://www.amd.com/es/products/software/adaptive-socs-and-fpgas/vitis/vitis-libraries/vitis-vision.html

[9] «Using Vitis Vision Libraries and OpenCV». Accedido: 18 de enero de 2025. [En línea]. Disponible en: https://adaptivesupport.amd.com/s/question/0D52E00006hpOJESA2/using-vitis-vision-libraries-and-opencv?language=en_US

[10] G. Wu, J. Yang, y H. Yang, «Real-time low-power binocular stereo vision based on FPGA», *J Real Time Image Process*, vol. 19, n.º 1, pp. 29-39, feb. 2022, doi: 10.1007/s11554-021-01158-z.

[11] «TUL». Accedido: 31 de enero de 2025. [En línea]. Disponible en: https://www.tulembedded.com/fpga/ProductsPYNQ-Z2.html

[12] «OV5640 Camera Board (C) - Waveshare Wiki». Accedido: 31 de enero de 2025. [En línea]. Disponible en: https://www.waveshare.com/wiki/OV5640_Camera_Board_(C)

[13] «APPLICATION NOTE O mni ision® OmniVision Serial Camera Control Bus (SCCB) Functional Specification Revision Number Date Revision».

[14] «SoC Zynq 7000». Accedido: 2 de mayo de 2025. [En línea]. Disponible en: https://www.amd.com/es/products/adaptive-socs-and-fpgas/soc/zynq-7000.html

[15] «AMBA® AXI Protocol Specification», 2003. [En línea]. Disponible en: http://www.arm.com/company/

[16] «Tul 撼 訊 科 技 ». Accedido: 8 de julio de 2025. [En línea]. Disponible en: https://www.tul.com.tw/en/News/NewsList

[17] Xilinx y Inc, «Zynq-7000 SoC Family Product Selection Guide», 2014.

[18] «OV5640 Auto FocusCamera Module Application Notes OV5640 Auto Focus Camera Module Application Notes (with DVP Interface)». Accedido: 11 de julio de 2025. [En línea]. Disponible en: https://blog.arducam.com/downloads/modules/OV5640/OV5640_Software_app_note_parallel.pdf

[19] «Rolling shutter - Wikipedia, la enciclopedia libre». Accedido: 21 de mayo de 2025. [En línea]. Disponible en: https://es.wikipedia.org/wiki/Rolling_shutter

[20] «OV5640- Camera board schematic». Accedido: 15 de mayo de 2025. [En línea]. Disponible en: https://files.waveshare.com/upload/1/1e/OV5640-Camera-Board-Schematic.pdf

[21] «OV5640 Camera Board (C) User Manual». Accedido: 11 de julio de 2025. [En línea]. Disponible en: https://download.kamami.pl/p1180728-OV5640_Camera_Board_%28C%29_User_Manual_EN.pdf

[22] «Vivado Design Suite - AMD / Xilinx | Mouser». Accedido: 24 de mayo de 2025. [En línea]. Disponible en: https://www.mouser.es/new/xilinx/amd-xilinx-vivado-design-suite/?srsltid=AfmBOoqMbWwoDUnHGRVC5Vd-vM3WCTbJjOrAE0YS_OxjcKnScxXl7sct

[23] «AMBA® AXI4 Interface Protocol». Accedido: 9 de julio de 2025. [En línea]. Disponible en: https://www.amd.com/es/products/adaptive-socs-and-fpgas/intellectual-property/axi.html

[24] Arm, «AMBA® AXI Protocol Specification», 2003. [En línea]. Disponible en: http://www.arm.com/company/

[25] «Introduction to AMBA AXI4», 2020. [En línea]. Disponible en: http://www.arm.com/company/policies/trademarks.

[26] «The hard part of building a bursting AXI Master». Accedido: 9 de julio de 2025. [En línea]. Disponible en: https://zipcpu.com/blog/2020/06/16/axiaddr-limits

[27] «1.2. Data Exchange». Accedido: 9 de julio de 2025. [En línea]. Disponible en: https://www.intel.com/content/www/us/en/docs/programmable/683397/current/data-exchange.html

[28] «Formatos YUV recomendados de 8 bits para la representación de vídeo - Win32 apps | Microsoft Learn». Accedido: 25 de mayo de 2025. [En línea]. Disponible en: https://learn.microsoft.com/es-es/windows/win32/medfound/recommended-8-bit-yuv-formats-for-video-rendering

[29] «Formatos de vídeo YUV de 10 y 16 bits - Win32 apps | Microsoft Learn». Accedido: 25 de mayo de 2025. [En línea]. Disponible en: https://learn.microsoft.com/es-es/windows/win32/medfound/10-bit-and-16-bit-yuv-video-formats

[30] «AXI4-Stream Subset Converter • AXI4-Stream Infrastructure IP Suite (PG085) • Reader • AMD Technical Information Portal». Accedido: 20 de junio de 2025. [En línea]. Disponible en: https://docs.amd.com/r/en-US/pg085-axi4stream-infrastructure/AXI4-Stream-Subset-Converter?tocId=VUGADTVnQ8nDgI86Et2EkA

[31] «Video Framebuffer Write - Xilinx Wiki - Confluence». Accedido: 23 de junio de 2025. [En línea]. Disponible en: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842236/Video+Framebuffer+Write

[32] «Features • Video Test Pattern Generator • Reader • AMD Technical Information Portal». Accedido: 23 de junio de 2025. [En línea]. Disponible en: https://docs.amd.com/r/en-US/pg103-v-tpg/Features

[33] «Video Timing Controller». Accedido: 23 de junio de 2025. [En línea]. Disponible en: https://www.amd.com/en/products/adaptive-socs-and-fpgas/intellectual-property/ef-di-vid-timing.html

[34] «RGB-to-DVI (Source) 1.4 IP Core User Guide». [En línea]. Disponible en: www.digilentinc.com

[35] «Generating the Bitstream • Vitis Tutorials: Embedded Software (XD260) • Reader • AMD Technical Information Portal». Accedido: 25 de junio de 2025. [En línea]. Disponible en: https://docs.amd.com/r/en-US/Vitis-Tutorials-Embedded-Software/Generating-the-Bitstream

[36] Xilinx publication- User guide, «Stereo Local Block Matching». [En línea]. Disponible en: https://github.com/Xilinx/Xilinx_Base_Runtime

[37] D. Scharstein y R. Szeliski, «A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms». [En línea]. Disponible en: www.middlebury.edu/stereo.

[38] R. Zabih y J. Woodfill, «Non-parametric local transforms for computing visual correspondence», *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 801 LNCS, pp. 151-158, 1994, doi: 10.1007/BFB0028345.

[39] N. Einecke y J. Eggert, «A multi-block-matching approach for stereo», *IEEE Intelligent Vehicles Symposium, Proceedings*, vol. 2015-August, pp. 585-592, ago. 2015, doi: 10.1109/IVS.2015.7225748.

[40] Q. Chang y T. Maruyama, «Real-Time Stereo Vision System: A Multi-Block Matching on GPU», *IEEE Access*, vol. 6, pp. 42030-42046, jul. 2018, doi: 10.1109/ACCESS.2018.2859445.

[41] «Vitis Vision Library». Accedido: 26 de junio de 2025. [En línea]. Disponible en: https://www.amd.com/es/products/software/adaptive-socs-and-fpgas/vitis/vitis-libraries/vitis-vision.html

[42] «HLS Pragmas • Vitis High-Level Synthesis User Guide (UG1399) • Reader • AMD Technical Information Portal». Accedido: 28 de junio de 2025. [En línea]. Disponible en: https://docs.amd.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas

[43] «Basic Features • Vitis Libraries • Reader • AMD Technical Information Portal». Accedido: 28 de junio de 2025. [En línea]. Disponible en: https://docs.amd.com/r/en-US/Vitis_Libraries/vision/overview.html_0_0

[44] «Evaluating the Accuracy of Single Camera Calibration». Accedido: 27 de junio de 2025. [En línea]. Disponible en: https://es.mathworks.com/help/releases/R2024b/vision/ug/evaluating-the-accuracy-of-single-camera-calibration.html

[45] José Javier Alcalde Sanz y Arturo de la Escalera Hueso, «Diseño de un Protocolo de Calibración de Cámaras Estéreo», 2014. Accedido: 27 de junio de 2025. [En línea]. Disponible en: https://e-archivo.uc3m.es/rest/api/core/bitstreams/26f68db5-091b-45b3-99d8-3ada9a6d08b5/content

[46] «Using the Stereo Camera Calibrator App». Accedido: 27 de junio de 2025. [En línea]. Disponible en: https://es.mathworks.com/help/releases/R2024b/vision/ug/using-the-stereo-camera-calibrator-app.html

[47] «OpenCV: Camera Calibration and 3D Reconstruction». Accedido: 19 de junio de 2025. [En línea]. Disponible en: https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html

[48] «Brown's Distortion Model | Use It To Compensate For Displacement of Pixels in Computer Vision & Solve Camera Distortion». Accedido: 26 de junio de 2025. [En línea]. Disponible en: https://www.foamcoreprint.com/blog/what-are-calibration-targets?srsltid=AfmBOorZH3wxBXTFZNaKlkmXi2Opextm2lmNqkBQJbjxCH1eae23x8en

[49] «OpenCV: Camera Calibration and 3D Reconstruction». Accedido: 28 de junio de 2025. [En línea]. Disponible en:

https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html#ga617b1685d4059c6040827800e72ad2b6

[50] «Difference of Gaussian Filter • Vitis Libraries • Reader • AMD Technical Information Portal». Accedido: 29 de junio de 2025. [En línea]. Disponible en: https://docs.amd.com/r/2023.2-English/Vitis_Libraries/vision/overview.html_3_4

[51] «Overview of Arbitrary Precision Integer Data Types • Vitis High-Level Synthesis User Guide (UG1399) • Reader • AMD Technical Information Portal». Accedido: 29 de junio de 2025. [En línea]. Disponible en: https://docs.amd.com/r/en-US/ug1399-vitis-hls/Overview-of-Arbitrary-Precision-Integer-Data-Types

[52] «Tipos de operaciones morfológicas - MATLAB & Simulink». Accedido: 29 de junio de 2025. [En línea]. Disponible en: https://es.mathworks.com/help/images/morphological-dilation-and-erosion.html

PLIEGO DE CONDICIONES

Esta sección detalla los términos y requisitos bajo los cuales se ha llevado a cabo este Trabajo Fin de Grado. Se describen los recursos *hardware* y *software* utilizados durante su desarrollo.

PC.1. RECURSOS *HARDWARE*

La Tabla PC.1 muestra los recursos hardware empleados para el desarrollo de este proyecto:

Equipo / Dispositivo	Modelo	Fabricante
Ordenador personal de sobremesa con conexión a internet	Intel Core i7-11700K @ 3.60 GHz 32 GB RAM (2667 MHz) NVIDIA GeForce RTX 3070 8 GB SSD NVMe 1 TB	Varios
Portátil personal con conexión a internet	ASUS X415	ASUS
Placa de desarrollo FPGA SoC	PYNQ-Z2 (Zynq-7020)	TUL / AMD Xilinx
Módulo de cámara estéreo	PZ-DOUBLE-OV5640-V1.1	Pú Zhì
Módulo de cámara estéreo	ATK-MC5640D-V1.91	HESUI
(Otros periféricos menores, cables, etc.)	–	Varios

Tabla PC.1. Recursos hardware

PC.2. RECURSOS *SOFTWARE*

La Tabla PC.2 recoge los elementos software empleados en el desarrollo de este proyecto:

Aplicación	Versión	Desarrollador / Comerciante
Sistema Operativo Windows	10 (64 bit) y 11 (64 bit)	Microsoft
Vivado Design Suite	2023.1	AMD/Xilinx
Vitis	2023.1	AMD/Xilinx
Vitis HLS	2023.1	AMD/Xilinx
Python (entorno científico)	3.11	Python Software Fdn.
MATLAB (licencia ULPGC)	R2024a	MathWorks
KiCad (CAE / PCB)	7.0	KiCad Project
OpenCV (librería visión)	4.10.0	OpenCV Org.
Vitis Vision Library	2023.1	AMD/Xilinx

Tabla PC.2. Recursos software

PRESUPUESTO

El presente anexo expone la estimación del presupuesto global del presente Trabajo Fin de Grado, separando los aspectos más significativos que lo componen, tales como la dedicación técnica del autor, los dispositivos y herramientas empleadas, tanto a nivel de hardware como de software, así como los costes asociados a la preparación y entrega del documento final. El objetivo de todo ello es el de ofrecer una justificación comprensible y completa del impacto económico que tuvo el desarrollo de este trabajo académico. Se evalúan los siguientes conceptos:

- Recursos humanos.
- Recursos materiales.
    - Recursos software.
    - Recursos hardware.
- Material fungible.
- Redacción del documento.
- Derechos de visado del Colegio Oficial de Ingenieros Técnicos de Telecomunicación (COITT).
- Gastos de tramitación y envío.
- Presupuesto final.

P.1. RECURSOS HUMANOS

Para cuantificar el coste asociado a esta dedicación, se ha tomado como referencia la retribución mensual asignada a un perfil técnico de Ingeniero Graduado en Telecomunicación, acorde con la titulación y responsabilidades asumidas en un TFG. En particular, según el acuerdo de retribuciones de la Universidad de Las Palmas de Gran Canaria (ULPGC) para personal técnico con titulación de Grado, se considera un salario de 760,05 € mensuales para una dedicación a tiempo parcial típica en este tipo de proyectos. Asumiendo un periodo de desarrollo de 4 meses (equivalente a unas 320 horas de trabajo, coherente con las 300 horas estimadas), el costo de recursos humanos queda determinado en la Tabla P.1.

Tipo de personal	Retribución mensual (€)	Duración	Coste Total (€)
Ingeniero técnico	760,05	4 meses	3040,20

Tabla P.1. Estimación de coste por recursos humanos

El coste resultante de la dedicación temporal del alumno asciende a TRES MIL CUARENTA EUROS CON VEINTE CÉNTIMOS, que representa el gasto imputable a recursos humanos (mano de obra técnica) del proyecto.

P.2. RECURSOS MATERIALES

En esta sección se encuentran los recursos materiales que son necesarios para el desarrollo del proyecto, tanto el equipamiento hardware como las herramientas software que se han utilizado. Para determinar el coste a imputar por el uso de dichos recursos es necesario amortizar el precio por el que se han adquirido a lo largo del periodo de vida útil, teniendo en cuenta únicamente la parte proporcional que se haya empleado en el trabajo. Para ello se emplea un modelo de amortización lineal, con una vida útil estimativa de 3 años (36 meses) para todos los componentes y un precio residual nulo al final de la vida útil de los mismos (hipótesis conservadora generalizada en entornos académicos). La fórmula que se adopta para la amortización anual de los componentes es la siguiente:

$$Cuota\ anual\ de\ amortización\ = \frac{Valor\ de\ adquisición - Valor\ residual}{Vida\ útil\ (años)}$$

Dividiendo después la cuota anual en meses y multiplicándola por los meses en uso efectivo en el TFG se obtiene el coste amortizado que se imputa al presupuesto del Trabajo Fin de Grado.

P.2.1. Recursos *hardware*

En la Tabla P.2 se detalla el coste correspondiente al *hardware* utilizado, que comprende tanto dispositivos físicos específicos del proyecto como el equipo informático general de soporte. Cada componente proporciona su valor de adquisición, la vida útil considerada (36 meses), la cuota de amortización mensual, así como el tiempo efectivo que se le ha encomendado en su desarrollo (en meses), obteniendo finalmente el importe prorrateado al proyecto:

Equipo	Valor de adquisición (€)	Vida útil	Cuota mensual (€)	Tiempo de uso	Importe (€)
Ordenador personal con conexión a Internet	1.500,00	36 meses	41,7	4 meses	166,7
Placa de desarrollo *PYNQ-Z2* (FPGA SoC Xilinx Zynq-7020)	240,00	36 meses	6,7	4 meses	26,7
Módulo de cámara estéreo PUZHI (PZ-DOUBLE-OV5640-V1.1))	55,56	36 meses	1,54	4 meses	6,17
Módulo de cámara estéreo HESUI (ATK-MC5640D-V1.91)	57,43	36 meses	1,60	4 meses	6,38
PCB	11,8	36 meses	0,32	4 meses	1,31
Total					**207,26**

Tabla P.2. Coste y amortización del material hardware utilizado

El coste total imputable a los recursos *hardware* asciende aproximadamente a DOSCIENTOS SIETE EUROS CON VEINTISEIS CÉNTIMOS.

P.2.1. Recursos *software*

El coste de las herramientas software utilizadas es descrito en la Tabla P.3, y se ha seguido el mismo criterio que el anteriormente utilizado. Como la mayoría de los programas utilizados son disponibles gratuitamente o son provistos bajo licencia académica, el coste amortizado se considera nulo.

Se ha considerado el sistema operativo Windows necesario para el desarrollo y redacción de documentos para la memoria, así como también las herramientas de desarrollo específicas de AMD/Xilinx.

Elemento software	Valor de adquisición (€)	Vida útil	Cuota mensual (€)	Tiempo de uso	Importe (€)
Sistema Operativo Windows 10	100,00	36 meses	2,8	4 meses	11,1
Entorno Xilinx Vivado 2023.1	0,00	36 meses	0,0	4 meses	0,0
Entorno Xilinx Vitis 2023.1	0,00	36 meses	0,0	4 meses	0,0
Python	0,00	36 meses	0,0	4 meses	0,0
MATLAB (licencia ULPGC*)	0,00	36 meses	0,0	4 meses	0,0
Entorno software KiCad	0,00	36 meses	0,0	4 meses	0,0
Librerías de visión e instalación de OpenCV	0,00	36 meses	0,0	4 meses	0,0
Total					**11,1**

Tabla P.3. Coste y amortización del material software utilizado

El coste final de los recursos hardware empleados es de ONCE EUROS CON DIEZ CÉNTIMOS.

P.3. MATERIAL FUNGIBLE

Además del hardware y software, es necesario considerar el consumo de material fungible durante la realización del TFG. Esto incluye principalmente los recursos para la preparación de la memoria impresa. En la Tabla P.4 se detallan estos costes:

Descripción	Coste (€)
Material de papelería	10,00
Impresión de la memoria	35,00
Encuadernación de ejemplares	5,00
Total	**50,00**

Tabla P.4. Costes estimados del material fungible

El coste total del material fungible es de CINCUENTA EUROS.

P.4. REDACCIÓN DEL DOCUMENTO

Con los costes base de mano de obra y materiales establecidos, a continuación, se calculan los honorarios de redacción, edición y maquetación de la memoria final del TFG. Para ello se aplica el criterio establecido por el Colegio Oficial de Ingenieros Técnicos de Telecomunicación (COITT), que especifica un porcentaje fijo del 7 % del presupuesto base, lo cual es modulado de acuerdo con un coeficiente según la escala del presupuesto. La expresión utilizada es la que se presenta a continuación:

$$R = 0{,}07 \times P \times C_n\,,$$

donde R es el coste originado por el trabajo de la redacción del documento, P es el presupuesto base (suma de recursos humanos y materiales) y C_n es un coeficiente de ponderación que depende del intervalo presupuestario que se encuentra P. En este caso, el presupuesto base es inferior a 30.050 €, por lo que el coeficiente se establece en Cn=1, según lo que marcan las tablas de tarifas vigentes del COITT.

Descripción	Coste (€)
Suma del trabajo tarifado	3.040,20
Amortización de los recursos *software*	11,1
Amortización de los recursos *hardware*	200,1
Total	**3.251,4**

Tabla P.5. Total de las amortizaciones y trabajo tarifado

Como se observa en la Tabla P.5 si se considera el presupuesto como la suma del trabajo tarifado (3.040,20 €) más la amortización de material *hardware* (200,1 €) y de *software* (11,1 €), se tiene P ≈ 3.251,4 €. Por tanto, aplicando la expresión utilizada anteriormente:

$$R = \ 0{,}07 \times 3.251{,}4 \times 1 \ \approx \ 227{,}60\,€$$

El coste derivado de la redacción del documento es de DOSCIENTOS VEINTISIETE EUROS CON SESENTA CÉNTIMOS.

P.5. DERECHOS DE VISADO DEL COITT

Para proceder al registro y visado oficial del proyecto ante el colegio oficial correspondiente, es necesario abonar la tasa de visado. Siguiendo las tarifas del COITT para proyectos técnicos de alcance general (ejercicio en curso), además de que el derecho de visado se calcula de la siguiente manera:

$$V = 0{,}006 \times P_1 \times C_1 + 0{,}003 \times P_2 \times C_2,$$

donde P_1 es el Presupuesto General del Proyecto (presupuesto base más honorarios de redacción), y C_1 es el coeficiente reductor que le corresponde a dicho presupuesto P_1. Por otro lado, P_2 es el presupuesto de ejecución material de obra (si hubiera) y C_2 su coeficiente reductor. En el presente TFG no existe obra civil asociada, por lo que $P_2 = 0$ y el segundo término de la ecuación queda anulado. Además, puesto que el presupuesto P_1 (suma de P + R) no supera los 30.050 €, representa que corresponde emplear la aplicación del coeficiente $C_1 = 1$ según las tarifas oficiales del COITT. Considerando esto, primero se determina el Presupuesto General del Proyecto: En este caso:

$$P\_1 = P + R \approx 3.251{,}4 + 227{,}6 = 3.479{,}0\,€$$

(presupuesto general del proyecto, sin impuestos).

Presentando la fórmula para la aplicación del visado:

$$V = 0{,}006 \times 3.479{,}0 \times 1 = 20{,}87\,€$$

Por ende, el coste correspondiente a los derechos de visado del TFG se estima en VEINTE EUROS Y OCHENTA Y SIETE CÉNTIMOS.

P.6. GASTOS DE TRAMITACIÓN Y ENVÍO

Cabe mencionar que, adicionalmente, el COITT contempla unos gastos administrativos de tramitación de los proyectos visados, para cubrir el registro, emisión de certificados y archivo. En el caso de tramitación telemática (visado electrónico), dicho gasto administrativo oscila entre 9,00 € y 12 €. Dado

que el visado de este TFG se realizaría de forma digital, se incluye esta cuantía fija en el presupuesto. Por tanto, los gastos de tramitación para este documento ascienden a NUEVE EUROS.

P.7. PRESUPUESTO FINAL DEL PROYECTO

Este Trabajo Fin de Grado está sujeto a la aplicación del Impuesto General Indirecto Canario (IGIC), que corresponde al 7 % del importe total del presupuesto. La Tabla P.6 muestra el resumen completo del coste final del proyecto.

Concepto	Importe (€)
Trabajo tarifado (recursos humanos)	3.040,20
Amortización material *hardware*	207,26
Amortización material *software*	11,10
Redacción, edición y maquetación del TFG	227,60
Derechos de visado (COITT)	20,90
Gastos administrativos de tramitación	9,00
Material fungible (impresión + encuadernación)	50,00
Total (sin IGIC)	3.566,06
IGIC (7%)	249,62
TOTAL	**3.815,68**

Tabla P.6. Presupuesto final

El presupuesto completo asociado a la realización del presente TFG es de TRES MIL OCHOCIENTOS QUINCE EUROS CON SESENTA Y OCHO CÉNTIMOS.

En Las Palmas de Gran Canaria, a 17 de Julio de 2025