Universidad de Alcalá
Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

Trabajo Fin de Máster

Codiseño de un sistema HW/SW para el procesamiento de vídeo en tiempo real

Autor: Rodrigo Llorente Aragón

Tutor: Alfredo Gardel Vicente

2019

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

Trabajo Fin de Máster

Codiseño de un sistema HW/SW para el procesamiento de vídeo en tiempo real

Autor: Rodrigo Llorente Aragón

Tutor: Alfredo Gardel Vicente

TRIBUNAL:

Presidente: Ignacio Bravo Muñoz

Vocal 1º: Óscar Rodríguez Polo

Vocal 2º: Alfredo Gardel Vicente

FECHA: 30 de septiembre de 2019

Resumen

En este trabajo se realiza un sistema de captura, procesado y visualización de imágenes en un dispositivo *SoC*. Este dispositivo está formado principalmente por un sistema de procesamiento *software* junto a una parte de *hardware* programable. Por ello, en la exploración de las alternativas de diseño se realiza un análisis de codiseño HW/SW, determinando las partes que son implementadas en *hardware* y cuales son ejecutadas en *software*. En el TFM se explica e implementa el algoritmo HOG que permite extraer información de las imágenes capturadas. Posteriormente, junto con un algoritmo de aprendizaje como las SVM, los descriptores HOG obtenidos de las imágenes pueden ser utilizados para detectar distintos elementos, como por ejemplo personas.

Palabras clave: SoC (*System on Chip*), Co-diseño HW/SW, MicroZed, HOG (Histograma de gradientes orientados), SVM (Máquinas vector soporte)

Abstract

In this work a system of capture, processing and visualization of images in a SoC platform is developed. This device consists mainly of a software processing system together with a part of programmable hardware. For that reason, in the study of the design alternatives an HW/SW co-design analysis is carried out, determining which parts are whether implemented in hardware or executed in software. The HOG algorithm that allows extracting information from the captured images is explained and deployed in this work. Finally, along with a learning algorithm such as SVM, the HOG descriptors retrieved from the images can be used to detect different elements, such as people.

Keywords: SoC (*System on Chip*), HW/SW Co-design, MicroZed, HOG (*Histogram of Oriented Gradients*), SVM (*Support Vector Machine*)

Resumen extendido

En la actualidad, las aplicaciones requieren obtener información del entorno que las rodea, de forma que el sistema pueda tomar decisiones y actuar sobre el entorno. Una de las formas más utilizadas para obtener dicha información es mediante el uso de fotografías o vídeos capturados por diferentes tipos de sensores de imagen. Sin embargo, el procesado de estas imágenes en tiempo real supone un alto coste computacional, lo que puede provocar cierto retardo en la obtención de la información.

Existen multitud de aplicaciones, como las relacionadas con los sistemas de transporte inteligente, que imponen que la información sea extraída de la forma más rápida posible, incluso en tiempo real, de forma que se pueda reaccionar rápidamente ante situaciones que así lo requieran.

Pensando en ese tipo de aplicaciones, en este trabajo se propone realizar el procesamiento de imágenes empleando un sistema *SoC* (*System on Chip*) que incluye en un único circuito integrado todos los elementos que necesitan para su funcionamiento. Algunos de estos dispositivos incluyen, además del microprocesador, la memoria y los interfaces de comunicación, una *FPGA* (*Field Programmable Gate Array)*. Estos dispositivos permiten implementar todo o una parte del algoritmo de procesamiento en este *hardware* reprogramable, logrando incrementar el rendimiento final del sistema, comparado con su ejecución completa en un microprocesador.

Este tipo de dispositivos requieren un codiseño *hardware-software*, en el cual se debe decidir que partes de la aplicación deben ejecutarse en la parte de *hardware* reconfigurable y cuales en el microprocesador.

En el caso de este trabajo, se emplea la tarjeta de desarrollo MicroZed cuyo elemento central es un *SoC* Zynq 7000 de Xilinx. Esta tarjeta dispone de diferentes interfaces de comunicación que le permiten comunicarse con otros dispositivos, obteniendo información de estos. Sin embargo, esta tarjeta no cuenta con las interfaces propias de un sistema de procesamiento de vídeo como, por ejemplo, un puerto HDMI. Para este tipo de aplicaciones, se puede acoplar a la tarjeta MicroZed un módulo de expansión denominado *MicroZed Embedded Vision Carrier Card* (EMBV) que dispone de algunos de los dispositivos que una aplicación de procesamiento de imágenes pueda necesitar como, por ejemplo, una interfaz HDMI. Además, este módulo de expansión permite acoplar diferentes sensores de imagen, lo que permite disponer de un sistema completo de captura, procesamiento y visualización de imágenes.

En este trabajo se explican detalladamente cada uno de los elementos que conforman el diseño *hardware* que permite la captura y visualización de las imágenes generadas por un sensor de imagen PYTHON 1300-C. Además, para el correcto funcionamiento del sistema, se comentan las distintas configuraciones *software* de alguno de los componentes que integran el diseño como, por ejemplo, el propio sensor de imagen o el dispositivo que genera la interfaz HDMI.

Una vez comprobado el correcto funcionamiento del diseño anterior, se puede introducir en este un primer procesado de las imágenes obtenidas. En este caso, se ha diseñado un sencillo algoritmo en HW que, introducido en el flujo de vídeo, calcula y visualiza sobre las imágenes capturadas el histograma de niveles de gris de estas.

Sin embargo, es posible realizar procesamientos más complejos que permitan extraer más información de las imágenes obtenidas. Uno de estos procesamientos son los descriptores visuales que describen algunas de las características observables de los elementos dispuestos en las imágenes o vídeos.

Uno de estos descriptores de características es el histograma de gradientes orientados (HOG) que se basa en la dirección del gradiente para describir los cambios de luminosidad en regiones locales de la imagen. Para ello, genera distintos histogramas a partir de la orientación de los gradientes entre píxeles vecinos, de forma que, cada uno de ellos recoge cuales son las direcciones predominantes en zonas reducidas denominadas *celdas*. De esta forma, un descriptor HOG da información local sobre los diferentes cambios de luminosidad en una imagen, permitiendo así conocer, incluso visualmente mediante su representación, la dirección de los bordes que definen los distintos objetos que se representan en ella.

La implementación del algoritmo HOG, tal y como está definido formalmente, supone un alto coste computacional, principalmente si este desea ser implementado en un dispositivo embebido, debido al elevado número de operaciones matemáticas que este emplea. Algunas de estas operaciones son el cálculo del arcotangente, la realización de potencias y raíces cuadradas y un gran número de multiplicaciones y divisiones. Además de lo anterior, este algoritmo supone un alto coste en memoria sobre todo si desea ser aplicado sobre una imagen completa.

Por todo lo anterior, en este trabajo se ha justificado la necesidad de simplificar este algoritmo, explicando detalladamente cada uno de los cálculos utilizados. Además, se ha justificado la validez de dicho algoritmo simplificado comprobando que sigue siendo válido para la extracción de características de una imagen.

Este algoritmo simplificado ha sido ejecutado en el procesador ARM del dispositivo Zynq utilizado, obteniendo unos tiempos de ejecución que pueden ser considerados demasiado elevados si se pretende realizar un procesamiento en tiempo real de las imágenes. Por ello, se ha adaptado dicho algoritmo para su implementación en la parte de *hardware* reconfigurable del dispositivo Zynq. Para ello, se ha utilizado la herramienta Vivado HLS que permite sintetizar a un lenguaje de descripción *hardware* un algoritmo codificado en un lenguaje de alto nivel como C. Cabe destacar que esta herramienta necesita que el programador conozca las principales limitaciones del desarrollo de un diseño HW de forma que este pueda decidir ciertos aspectos relacionados con la forma en que la síntesis es llevada a cabo.

Una vez realizado y validado el diseño HW, este ha sido ejecutado sobre la tarjeta de desarrollo y se ha comprobado su correcto funcionamiento, observando que se consigue un menor tiempo de ejecución que cuando este es ejecutado a nivel *software* en el microprocesador ARM.

Por último, se ha desarrollado una aplicación de visión artificial combinando los distintos desarrollos explicados a lo largo de esta memoria. Esta aplicación consiste en la detección de personas en las imágenes capturadas por el sensor PYTHON. Para ello, se hace uso de los descriptores calculados a partir del algoritmo HOG simplificado y de un algoritmo de aprendizaje supervisado como son las máquinas de vector soporte (SVM).

Para que el algoritmo de clasificación SVM pueda decidir sobre la existencia o no de personas a partir de un descriptor HOG, es necesario que este sea previamente entrenado a partir de observaciones ya etiquetadas. Tras el entrenamiento, se comprueba la validez de la función clasificadora con otras observaciones diferentes también previamente

etiquetadas. Una vez esta función ha sido validada, se incorpora al diseño y se comparan los resultados obtenidos tras su ejecución en *software* con los logrados cuando se implementa en *hardware*.

Así, la aplicación desarrollada emplea las imágenes capturadas por el sensor PYTHON para obtener los descriptores HOG y, a partir de estos, clasificar las distintas ventanas de detección empleando la función discriminante obtenida tras el entrenamiento SVM. Por último, una vez clasificadas como persona, estas ventanas son señaladas sobre la imagen visualizada en un monitor HDMI.

Índice general

Índice figuras

Capítulo 1: Introducción

1.1 Contexto

En la actualidad, una gran cantidad de aplicaciones requieren obtener información del entorno que las rodea, de forma que dicha información pueda emplearse para decidir qué acciones deben llevarse a cabo en un determinado momento.

Una de las técnicas para extraer datos del entorno es mediante el empleo de imágenes capturadas por distintos tipos de sensores (RGB, profundidad, térmicas, hiper-espectrales, etc) o, incluso, secuencias de video, con el limitante de que a menudo deben ser procesadas en tiempo real. En este TFM se entiende que un algoritmo se ejecuta en tiempo real si el tiempo de ejecución es inferior al tiempo máximo de respuesta necesario para la aplicación final a la que se destina el sistema. En general, este procesamiento sobre un flujo de datos provenientes de un sensor de imagen tiene un alto consumo computacional, lo que provoca un cierto retardo en la obtención de la información. De esta forma, no contemplar los requisitos de tiempo de ejecución puede dar lugar a una pérdida de información importante.

En determinadas aplicaciones, como por ejemplo en las relacionadas con los sistemas inteligentes de transporte, es vital que la información sea obtenida de la forma más rápida posible, permitiendo disponer de un mayor tiempo de actuación para responder a dicha información. Un ejemplo más concreto de esto puede tomarse de los sistemas de conducción autónoma en los que, entre muchos otros dispositivos, una cámara captura imágenes de la carretera y sus zonas aledañas. Estas imágenes deben ser procesadas en tiempo real de forma que para cada imagen a procesar se detecten los diferentes objetos del entorno como, por ejemplo, distintos tipos de vehículos u obstáculos. Además, una vez detectados se deben analizar las posibles reacciones a tomar y, finalmente si es necesario, actuar sobre los sistemas del automóvil. En este análisis se ha obviado el número de imágenes a procesar por segundo. Por un lado, existe una restricción de latencia máxima para el procesamiento de una imagen y, por otro lado, la cadencia o número de imágenes que se pueden procesar por segundo. Los sistemas de cómputo en paralelo (por ejemplo los supercomputadores) permiten procesar un gran número de imágenes por segundo pero el lograr una latencia reducida necesita, en general, tener sistemas de procesamiento cercanos al sensor.

Este tipo de tareas con latencia baja son realizadas por sistemas denominados empotrados que, a diferencia de los ordenadores de propósito general, se diseñan con el objetivo de realizar un conjunto de tareas específicas. Encontrar el balance entre la especificidad necesaria y la flexibilidad de programación del sistema es una tarea ardua que el diseñador debe afrontar en los momentos iniciales del desarrollo.

Aprovechando los avances en las últimas tecnologías electrónicos, en este TFM se realizará el diseño de un sistema empotrado haciendo uso de lo que se denomina *System on Chip* (*SoC*). Se engloban dentro de la categoría *SoC* a los dispositivos electrónicos que disponen dentro de un único circuito integrado multitud de los elementos que serán necesarios para el funcionamiento de un sistema de procesamiento empotrado. Algunos de los elementos más comunes que constituyen un *SoC* son: un microprocesador, módulos de memoria de distintos tipos (ROM, RAM, etc.), osciladores y diferentes controladores para la comunicación con el exterior.

Actualmente existen *SoC* aún más complejos que, además de los componentes anteriores, incluyen una *FPGA* o *Field Programmable Gate Array*. Estos dispositivos reprogramables

están conformados por bloques de lógica configurables que pueden ser interconectados para implementar una determinada función desarrollada mediante un lenguaje de descripción *hardware* de bajo nivel como VHDL/Verilog o de alto nivel como C-HLS.

El diseño de una aplicación para este tipo de sistemas debe ser realizado de forma dual, determinando qué partes de la aplicación son implementadas en la parte de *hardware* reconfigurable y cuáles son ejecutadas en la parte *software*. Por ello, estas técnicas de diseño son denominadas co-diseño HW/SW.

Dado que en la titulación se ha podido trabajar con circuitos electrónicos programables de Xilinx, en este TFM se va a hacer uso de un *SoC* Zynq desarrollado por Xilinx y conformado por una parte *software* cuyo elemento central es un procesador ARM de doble núcleo al que se acopla una parte de lógica reprogramable equivalente a una FPGA Artix-7.

El diseño de todas las aplicaciones que se van a desarrollar a lo largo de este trabajo se realiza utilizando el conjunto de herramientas que el fabricante ofrece para ello. Todas estas herramientas se agrupan bajo el nombre *Xilinx Vivado Design Suite* que incluye, principalmente, 3 programas: *Vivado*, *Vivado Software Development Kit* (SDK) y *Vivado High Level Synthesis* (HLS).

Dado el gran interés de los desarrolladores digitales por los sistemas empotrados para el procesamiento de imágenes, existen tarjetas de desarrollo que incorporan este tipo de dispositivos *SoC* junto a una tarjeta de captura de imágenes con los buses de interconexión ya definidos.

1.2 Objetivos

El objetivo principal del presente TFM es el estudio y puesta en marcha de un sistema de procesado de imágenes en un sistema compacto con unas ciertas restricciones de tiempo real. El campo de aplicación es la visión artificial, con especial interés en su utilización en sistemas inteligentes de transporte, que permiten conocer información acerca del entorno mediante la captura de imágenes y su posterior procesado para, por ejemplo, detectar personas en un área determinada.

Una vez definido el objetivo principal, se extraen diferentes objetivos parciales (más concretos) que se obtendrán con la realización del proyecto:

- Estudio y puesta en funcionamiento de la tarjeta de desarrollo MicroZed junto con el módulo de expansión *MicroZed Embedded Vision Carrier Card* y el sensor de imagen *PYTHON-1300-C Camera Module*.
- Desarrollo de un sistema de captura, procesado y visualización de imágenes en tiempo real: justificación del diseño, conexionado y configuración.
- Desarrollo y adición al diseño anterior de una sencilla aplicación para el procesamiento de las imágenes obtenidas: cálculo y visualización del histograma de niveles de gris.
- Estudio de un algoritmo de procesamiento de imágenes complejo como es el Histograma de Gradientes Orientados (HOG): justificación e implementación, en *software* y en *hardware*, de una simplificación de dicho algoritmo.
- Desarrollo de una aplicación de visión artificial para el procesamiento de vídeo: detección de personas a partir de los descriptores HOG y un clasificador SVM. Entrenamiento y validación de una máquina vector soporte. Implementación de la

función discriminante tanto a nivel *software* como a nivel *hardware* y comparación de resultados.
- Comparación de distintas alternativas HW/SW para procesamiento de datos (imágenes) en tiempo real y procedimiento de validación de los resultados frente a los obtenidos en simulación.

1.3 Estructura de la memoria

En este apartado se realiza una descripción del resto de capítulos desarrollados en el presente TFM.

En el siguiente capítulo se describen los distintos dispositivos utilizados a lo largo del TFM: la tarjeta de desarrollo MicroZed, el módulo de expansión *MicroZed Embedded Visión Carrier Card* (EMBV) y el módulo con el sensor de imagen *ON Semiconductor PYTHON 1300-C Camera Module*.

En el capítulo 3 se describe detalladamente una aplicación que, haciendo uso de los dispositivos anteriormente explicados, captura, procesa y visualiza las imágenes capturadas por el sensor PYTHON. Para ello, primero se explica el diseño HW necesario, centrándonos en los distintos bloques IP utilizados y su configuración. Posteriormente se comentan las configuraciones SW necesarias para el correcto funcionamiento de los distintos dispositivos que incluye la tarjeta de desarrollo y los módulos de expansión. Por último, se ejecuta la aplicación, indicando el conexionado necesario y comentando los resultados obtenidos.

En el capítulo 4 se implementa y añade al diseño anterior un ejemplo de procesado HW en tiempo real del flujo de vídeo. Este procesado será el cálculo y la visualización del histograma de niveles de gris de cada una de las imágenes capturadas por el sensor de imagen. La codificación del algoritmo se llevará a cabo usando la herramienta Vivado HLS que permite describir en lenguaje de alto nivel un determinado algoritmo y posteriormente sintetizarlo para generar un bloque HW implementable.

En el capítulo 5 se estudia en profundidad el descriptor de características HOG, indicando sus principales parámetros y características. Se implementa dicho algoritmo en SW y se justifica la necesidad de simplificar su cálculo, indicando las variaciones introducidas y comparando los resultados entre ambos. Este algoritmo HOG simplificado es implementado tanto a nivel SW como HW, comparando los resultados obtenidos.

Se termina el desarrollo del TFM con el capítulo 6 donde se muestra una aplicación de visión artificial para la detección de personas en imágenes, combinando el diseño del capítulo 3, el algoritmo HOG simplificado implementado en el capítulo 5 y un algoritmo de clasificación como son las máquinas de vector soporte. El uso de una SVM supone un primer paso de entrenamiento y validación, tras el cual se puede decidir sobre la existencia o no de una persona aplicando la función discriminante obtenida. La aplicación de esta función discriminante se ha ejecutado tanto a nivel *software* como a nivel *hardware*, comparando ambos resultados.

La memoria técnica finaliza con la extracción de las principales conclusiones del TFM y propuesta de varias líneas de trabajo futuras.

Capítulo 2: Dispositivos utilizados

En los siguientes apartados se detallan los 3 dispositivos *hardware* que se han utilizado a lo largo de este trabajo:

1. Tarjeta Microzed basada en el dispositivo Zynq-7000 *All Programmable SoC* de Xilinx.
2. Módulo de expansión *MicroZed Embedded Vision Carrier Card* (EMBV) con dispositivos y conectores para el desarrollo de aplicaciones basadas en el procesamiento de imágenes.
3. Tarjeta con el sensor de imagen PYTHON 1300-C de ON Semiconductor e interfaz de conexión PCI Express.

2.1 Tarjeta de desarrollo MicroZed

La tarjeta de desarrollo Microzed es un dispositivo de bajo coste basado en el *chip* z7020 de la familia Zynq-7000 de Xilinx. Dispone, además, de una serie de conexiones e interfaces de comunicación que facilitan la incorporación de periféricos a la aplicación que se desea desarrollar.

2.1.1 Dispositivo Zynq-7000

El dispositivo Zynq, cuyo esquema se muestra en la Figura 2.1, es denominado *All Programmable SoC* ya que integra una FPGA de la serie 7 de 28 nm acoplada a un procesador ARM Dual-Core Cortex-A9. Por ello, el sistema puede ser dividido en dos partes claramente diferenciadas: parte de procesamiento (*Processing System* o PS) y parte de lógica programable (*Programmable Logic* o PL).

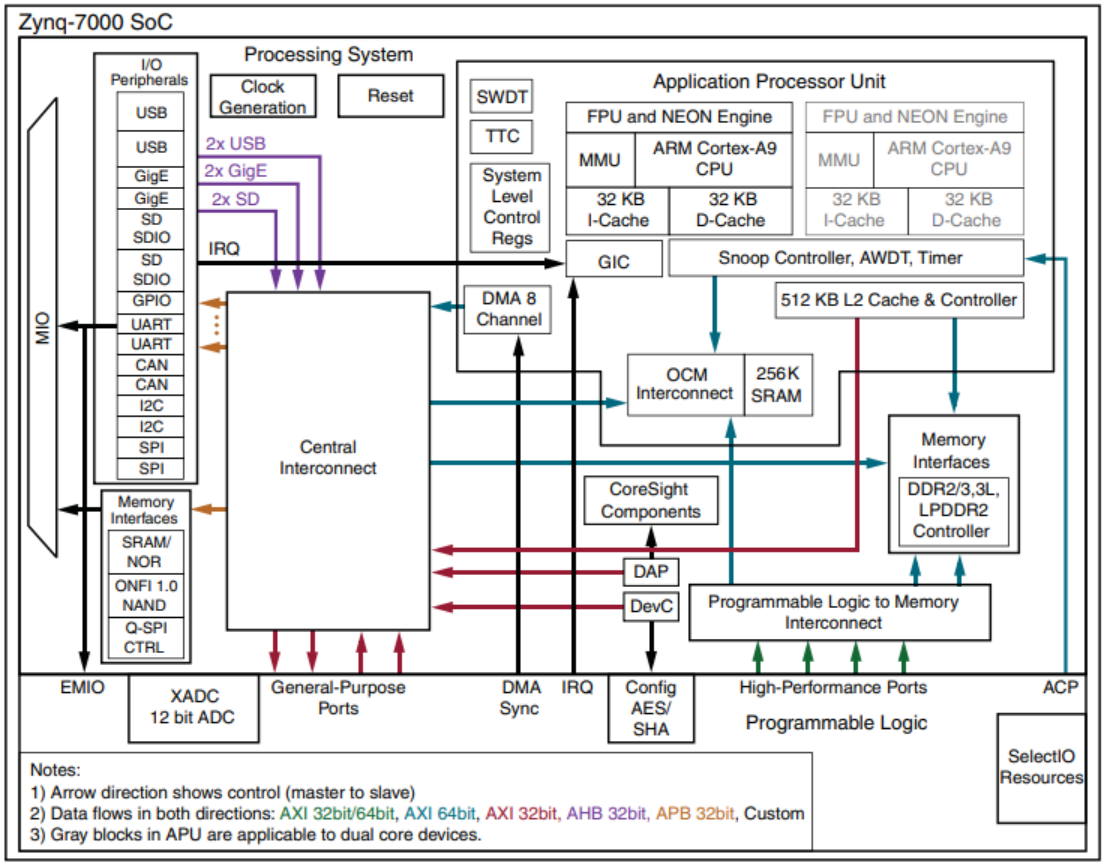


Figura 2.1: Diagrama Zynq-7000 [1]

El sistema de procesamiento está formado principalmente por la APU o *Application Processor Unit* en la cual se encuentran los dos núcleos ARM Cortex-A9. También se dispone de una memoria caché de 512 KB, una memoria RAM de 512 KB y un bloque de memoria ROM de 128 KB utilizada para el arranque. En el sistema de procesamiento también se encuentra la controladora DMA (*Direct Memory Access*) que permite realizar transferencias desde o hacia la memoria sin intervención del procesador.

En el sistema de procesamiento también se incluyen una serie de periféricos de entrada y salida que permiten la comunicación del dispositivo Zynq con otros sistemas. Los controladores disponibles son los siguientes: SPI, I2C, CAN, UART, GPIO, SDIO, USB y Ethernet Gigabit. Estos controladores están conectados con el exterior a través de la interfaz MIO (*Multiplexed I/O*).

El dispositivo Zynq dispone de distintos tipos de controladores que se encargan de la gestión de los diferentes tipos de memoria disponibles en la tarjeta de desarrollo. Por un lado, el controlador de memoria estática (*SMC*) que se encarga de la gestión de las memorias NOR y NAND Flash y el controlador Quad-SPI Flash que permite el control de este tipo de memoria no volátil ideal para el almacenamiento del código y datos de la aplicación. La tarjeta MicroZed dispone de una memoria Quad-SPI Flash de 128 Mbit de capacidad que soporta una tasa máxima de transferencia de 400 Mbps. Por otro lado, el control de la memoria dinámica se encarga de la gestión de las memorias DDR. La tarjeta de desarrollo MicroZed dispone de dos memorias DDR3 de Micron que suponen un total de 1 GB de memoria RAM.

Los distintos componentes del sistema de procesamiento están intercomunicados entre sí con buses de datos con arquitectura AMBA, tecnología desarrollada por ARM.

La parte lógica programable está conformada por una FPGA Artix-7 formada por 85000 células lógicas, 53200 LUTs, 106400 biestables, 4,9 MB de RAM y 220 DSP *Slices*.

Para conectar las dos partes que componen un dispositivo Zynq se emplea el protocolo AXI (*Advanced eXtensible Interface*) que forma parte de la especificación AMBA.

2.1.2 Conexiones en la tarjeta MicroZed

Las diferentes conexiones disponibles en la tarjeta MicroZed se muestran en la Figura 2.2.

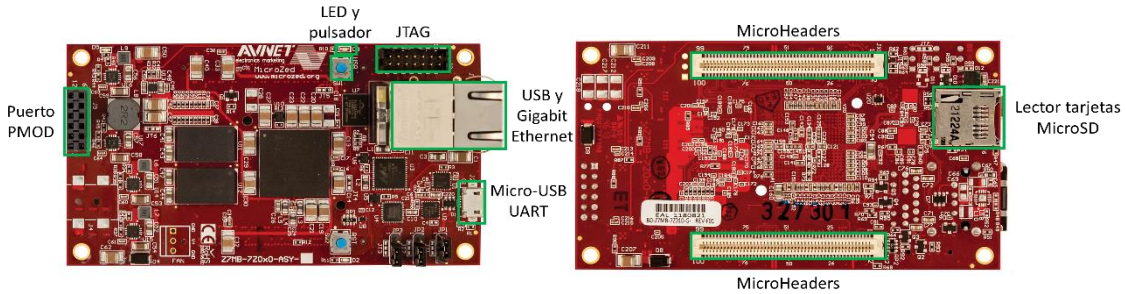


Figura 2.2: Conexiones tarjeta MicroZed

A continuación se enumeran cada una de estas conexiones, indicando brevemente sus principales características:

- Conectores *MicroHeaders*: Utilizados para conectar tarjetas de expansión que amplíen el número de interfaces y dispositivos accesibles desde el dispositivo Zynq.

- Lector tarjetas MicroSD: Conectado al controlador SD0 de la parte PS del dispositivo Zynq.
- USB UART: Conector micro-USB que permite la comunicación a través del puerto serie. Se implementa mediante el dispositivo CP2104 de Silicon Labs. Conectado al controlador UART1 del dispositivo Zynq.
- USB: Conectado al controlador USB0 del dispositivo Zynq.
- Gigabit Ethernet: Se dispone de un conector de tipo RJ-45 y se implementa mediante un *chip* Marvell 88E1512 PHY. Conectado al controlador ENET0 de la parte PS del dispositivo Zynq.
- Puerto genérico PMOD: Interfaz desarrollada por Digilent para la conexión de periféricos en FPGAs y microcontroladores. Dispone de 12 pines de conexión, 4 de alimentación, 3,3 voltios y tierra, y 8 configurables como entradas o salidas (GPIO). Se encuentra conectado a la parte PS y tiene asignados los puertos MIO 0 y del 9 al 15.
- LED.
- Pulsador.
- Conector JTAG.

Un diagrama con los distintos elementos disponibles y sus conexiones en la tarjeta de desarrollo MicroZed puede observarse en la Figura 2.3.

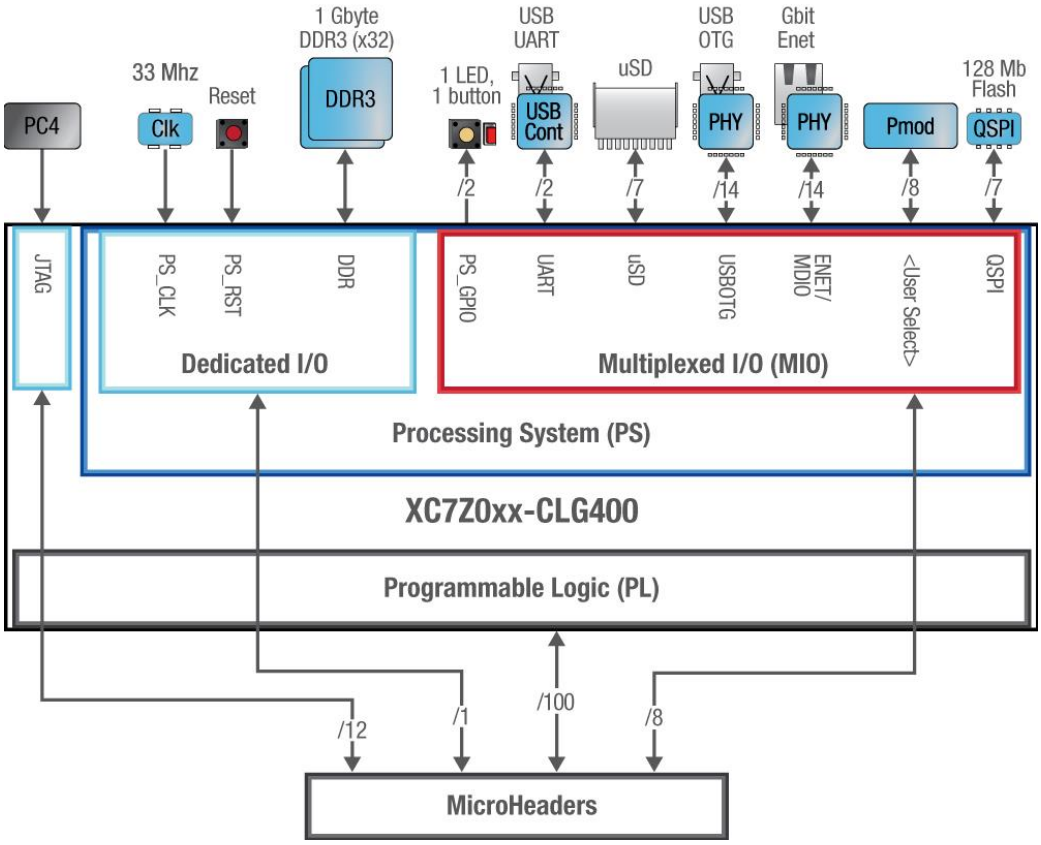


Figura 2.3: Diagrama de bloques tarjeta MicroZed [2]

Cabe destacar que el puerto PMOD y el conector JTAG se encuentran directamente conectados a alguno de los pines disponibles en los conectores MicroHeader, tal y como se puede observar en la Figura 2.3. Esto supone que, si se dispone de una tarjeta de expansión conectada, estas señales no pueden ser utilizadas simultáneamente desde la tarjeta MicroZed y desde la tarjeta de expansión.

2.2 MicroZed Embedded Vision Carrier Card (EMBV)

La tarjeta de expansión *MicroZed Embedded Vision Carrier Card* o *EMBV* provee de una serie de periféricos ideados para facilitar el desarrollo de aplicaciones basadas en procesamiento de imágenes en tiempo real. La tarjeta MicroZed se conecta a esta tarjeta de expansión a través de 2 conectores *MicroHeaders* con 100 pines cada uno.

Las principales interfaces de la tarjeta EMBV se pueden ver en la Figura 2.4 y son las siguientes:

- Interfaz de conexión para acoplar un sensor de imagen: Detallado en el apartado 2.2.2 de este documento.
- Entrada Micro-HDMI: Implementado mediante un dispositivo ADV7611 de Analog Devices. Únicamente soporta el formato YCbCr 4:2:2 con el objetivo de reducir el número de pines ocupados en el dispositivo Zynq.
- Salida Micro-HDMI: Detallado en el apartado 2.2.3 de este documento.
- Interfaz de Audio: Basado en el dispositivo ADAU1761 de Analog Devices permite la conexión de auriculares y micrófono a partir de conexiones tipo *jack*. Este dispositivo es un códec de audio estéreo de baja potencia que admite la grabación y reproducción estéreo en un rango desde 8 kHz a 96 kHz.
- Puerto Gigabit Ethernet: Implementado mediante el dispositivo 88E1512 PHY de Marvell.
- 3 puertos genéricos de tipo PMOD.
- 2xPulsadores.
- 2xLEDs.

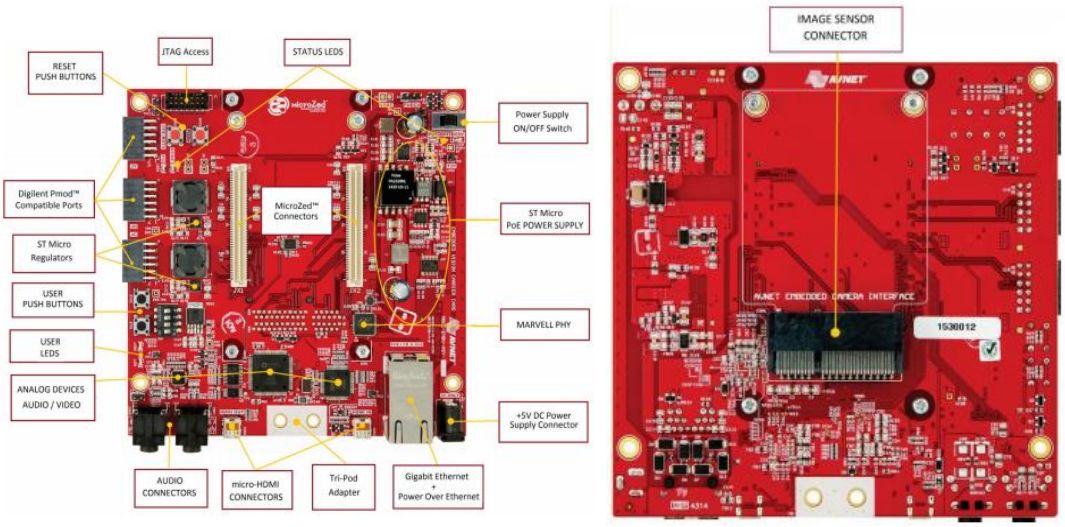


Figura 2.4: Tarjeta Embedded Vision [3]

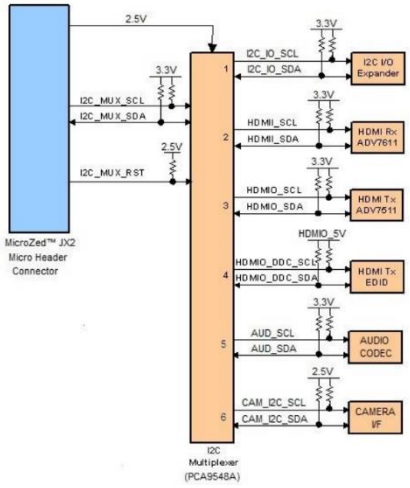
En los siguientes apartados se comentan más detalladamente las conexiones y dispositivos que van a ser utilizados posteriormente en este trabajo para el desarrollo de las diversas aplicaciones.

2.2.1 Comunicaciones y puertos I2C en la tarjeta EMBV

La tarjeta EMBV dispone de una gran cantidad de dispositivos e interfaces que soportan el protocolo I2C: *I2C IO Expander*, entrada HDMI, salida HDMI, HDMI DDC, códec de audio y la interfaz de la cámara.

La comunicación entre los distintos dispositivos de la tarjeta EMBV y el dispositivo Zynq-7000 se realiza a través de un multiplexor Texas Instruments PCA9548A. Este dispositivo dispone de 8 canales de salida I2C aunque únicamente se utilizan 6 en esta tarjeta. El control de este dispositivo se realiza desde un maestro I2C ubicado en la parte de procesamiento del dispositivo Zynq-7000.

El conexionado del multiplexor I2C y los dispositivos de la tarjeta EMBV conectados a cada uno de los puertos, pueden observarse en la Figura 2.5.

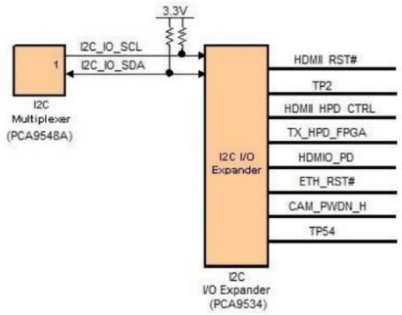


SW1 Setting	I/O Connection
I2C Multiplexer	0xE0 (PCA9548)
Mux Port 1	
I2C I/O Expander	0x40 (PCA9534)
Mux Port 2	
HDMI Input	0x98 (ADV7611) *
Mux Port 3	
HDMI Output	0x72 (ADV7511) *
Mux Port 4	
HDMI Output DDC EDID	0xA0
Mux Port 5	
Audio Codec	0x74 or 0x76 (ADAU1761) **
Mux Port 6	
Camera Interface	Module Dependent ***

Figura 2.5: Multiplexor I2C PCA9548A [3]

Con el objetivo de disponer de más señales de control de entrada/salida, se dispone de un expansor I2C conectado a algunas de las señales de control de los periféricos. Este dispositivo expansor PCA9534 de Texas Instruments, decodifica una de las interfaces del multiplexor generando 8 señales GPIO a su salida.

Cada una de las salidas del expansor se encuentra conectada a una de las señales de control de la interfaz, tal y como se puede apreciar en la Figura 2.6.



Control Signal	Function	Default Value	Direction	Expander Port
HDMII_RST#	HDMI Input Reset	Pulled-High	Output	P0
TP2	Test Point – 2	Open	I/O	P1
HDMII_HPD_CTRL	HDMI Input Hot Plug Detect Control	Pulled-Low	Output	P2
TX_HPD_FPGA	HDMI Output Hot Plug Detect	Pulled-Low	Input	P3
HDMIO_PD	HDMI Output Power Down	Pulled-Low	Output	P4
ETH_RST#	Gigabit Ethernet Reset	Pulled-High	Output	P5
CAM_PWDN_H	Camera Interface Power Down	Pulled-High	Output	P6
TP54	Test Point – 54	Open	I/O	P7

Figura 2.6: Expansor I2C PCA9534 [3]

2.2.2 Puerto de conexión del sensor de imagen (Camera Interface)

La tarjeta EMBV permite conectar diferentes modelos de sensores de imagen al conector PCI Express (PCIe x4) del que dispone.

Esta tarjeta provee 16 señales de configuración y control entre las que se encuentran los protocolos I2C y SPI, señal de *reset*, señal de reloj, entre otras. A excepción de las dos señales del protocolo I2C, conectadas al puerto 6 del multiplexor, y la señal *Camera Interface Power Down*, conectada al puerto 6 del expansor, el resto de señales se encuentran directamente conectadas a uno de los bancos alimentado a 2,5 voltios del dispositivo Zynq.

En relación al protocolo I2C y puesto que la tarjeta EMBV emplea un dispositivo multiplexor I2C configurado con la dirección *0xE0*, es necesario asegurarse de que el módulo de cámara que se vaya a utilizar no tenga un dispositivo I2C con esa misma dirección asignada. Además, cabe destacar que la tarjeta EMBV implementa las resistencias de *pull-up* del bus I2C, tal y como se puede observar en la Figura 2.7, por lo que también es necesario asegurar que el módulo de la cámara no los implemente.

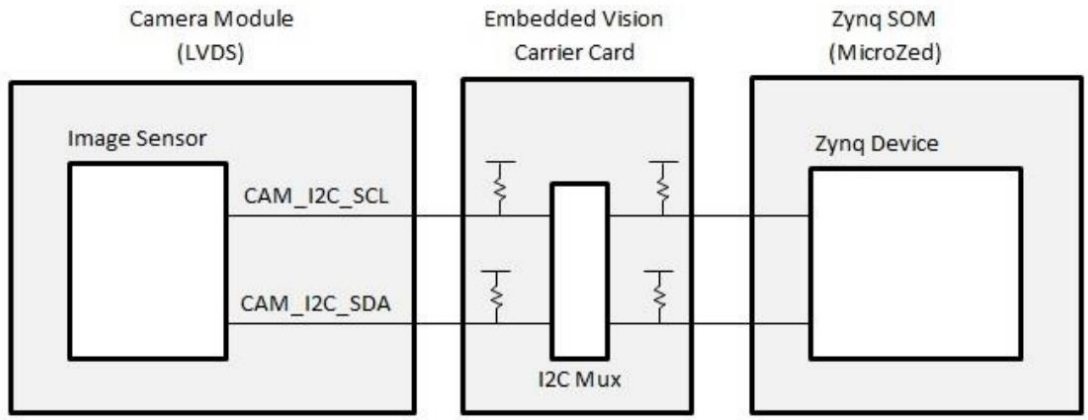


Figura 2.7: Resistencias pull-up del bus I2C [3]

En algunos escenarios y aplicaciones puede ser necesario iluminar el entorno para poder obtener imágenes con cierta calidad. Por ello, la interfaz provee dos señales de usuario denominadas *CAM_TORCH_EN* y *CAM_TORCH_PWM* que se podrían emplear para habilitar o deshabilitar una iluminación LED adicional que tuviera la placa del sensor óptico y para regular su intensidad empleando una señal PWM.

Además de las señales anteriores, se dispone de 3 señales denominadas *CAM_TRIGGER* que permiten generar señales de disparo hacia el módulo de la cámara. Si el sensor de imagen admite señales de disparo externas, alguna de ellas puede ser utilizada para la implementación de una iluminación LED de tipo *flash*.

Para la transmisión de los datos de video se proveen 10 señales diferenciales, 8 disponibles para datos, una señal de reloj y una señal de sincronismo. Los módulos pueden emplear cualquiera de los siguientes estándares: LVDS, Sub-LVDS, HiSPI y MIPI CSI-2. Sin embargo, el módulo EMBV no dispone de ningún circuito espacial para la generación o tratamiento de estas señales estando estas directamente conectadas a la tarjeta MicroZed. De esta forma, todos los circuitos específicos para el empleo de estos estándares deben encontrarse en el módulo de la cámara.

2.2.3 HDMI de salida

La interfaz HDMI de salida está implementada con el dispositivo ADV7511 de Analog Devices [4]. Aunque este dispositivo permite múltiples formatos de entrada y de salida, con el objetivo de reducir el número de pines de entrada y salida necesarios en la tarjeta MicroZed, se permite únicamente el formato de entrada YCbCr 4:2:2. Este formato es ampliamente utilizado y permite reducir el número de líneas necesarias de 24 a 16, submuestreando las componentes de color de las imágenes.

El conexionado entre el dispositivo ADV7511, la tarjeta MicroZed y el conector micro-HDMI de la tarjeta EMBV se puede observar en la Figura 2.8:

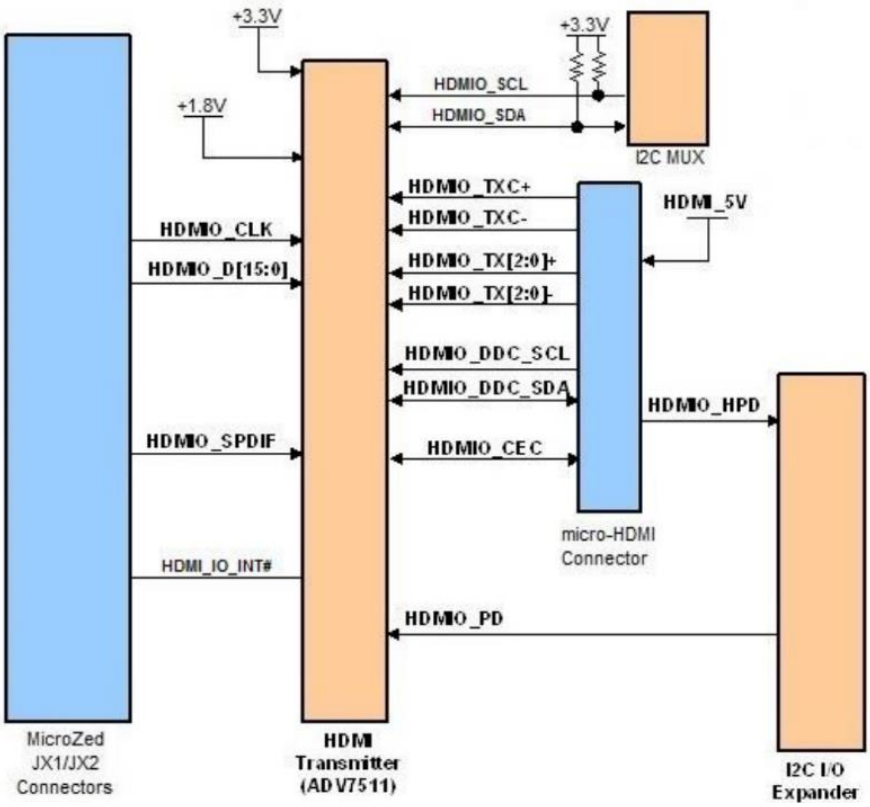


Figura 2.8: Conexionado dispositivo ADV7511, tarjeta MicroZed y conector micro-HDMI [3]

La inicialización y configuración de este dispositivo se realiza empleando el protocolo I2C, implementado tal y como se ha comentado anteriormente. De esta forma, en el puerto 3 del multiplexor I2C se encuentran las dos líneas de este protocolo (*HDMIO_SCL* y *HDMIO_SDA*) que permiten la conexión I2C con el dispositivo Zynq en la tarjeta MicroZed.

La línea *HDMIO_PD* proveniente del dispositivo expansor I2C se emplea para habilitar o deshabilitar el dispositivo ADV7511 que controla el puerto HDMI. Este pin se emplea, además, para configurar la dirección I2C del dispositivo: si está a nivel alto, la dirección será la 0x7A y, si está a nivel bajo, la 0x72 [5].

La señal *Hot Plug Detect* (*HDMIO_HPD*) del conector micro-HDMI está conectada al pin 3 del expansor I2C lo que permite conocer si hay algún dispositivo conectado a la interfaz HDMI de salida.

Los datos de entrada del transmisor HDMI se reciben de la tarjeta MicroZed a través de un conjunto de 16 señales denominadas *HDMIO_D[15:0]*. De esta forma, al emplear el formato

YCbCr 4:2:2, 8 de estas señales se emplean para el envío de los valores de luminancia (canal Y) de cada píxel y los 8 restantes para los valores de crominancia (canales diferencia de azul (Cb) y diferencia de rojo (Cr)). Posteriormente en este documento se detallará más pormenorizadamente el funcionamiento de este formato de imagen.

Asociada al bus de datos, la señal de reloj *HDMIO_CLK* indica la frecuencia de recepción de cada uno de los píxeles que compone la señal de video a transmitir por HDMI.

Las principales señales a la salida del dispositivo ADV7511 son de tipo diferencial según marca el protocolo HDMI. Así, se emplean 3 señales para la transmisión de datos (*HDMIO_TX[2:0]+* y *HDMIO_TX[2:0]-*) y una señal de reloj (*HDMIO_TXC+* y *HDMIO_TXC-*).

2.3 ON Semiconductor PYTHON 1300-C Camera Module

Esta tarjeta tiene como elemento central un sensor óptico PYTHON-1300-C de ON Semiconductor. Este sensor de tipo CMOS genera imágenes en color de 1280 por 1024 píxeles empleando, por tanto, una resolución *SXGA* o *Super Extended Graphics Array*.

En la Figura 2.9 se pueden apreciar los principales elementos que componen el módulo:

- Sensor PYTHON 1300.
- Interfaz PCIe x4: 64 pines de conexión, repartidos 32 en cada lado.
- Adaptador de tensión.
- Reguladores de tensión.
- Expansor I/O I2C.

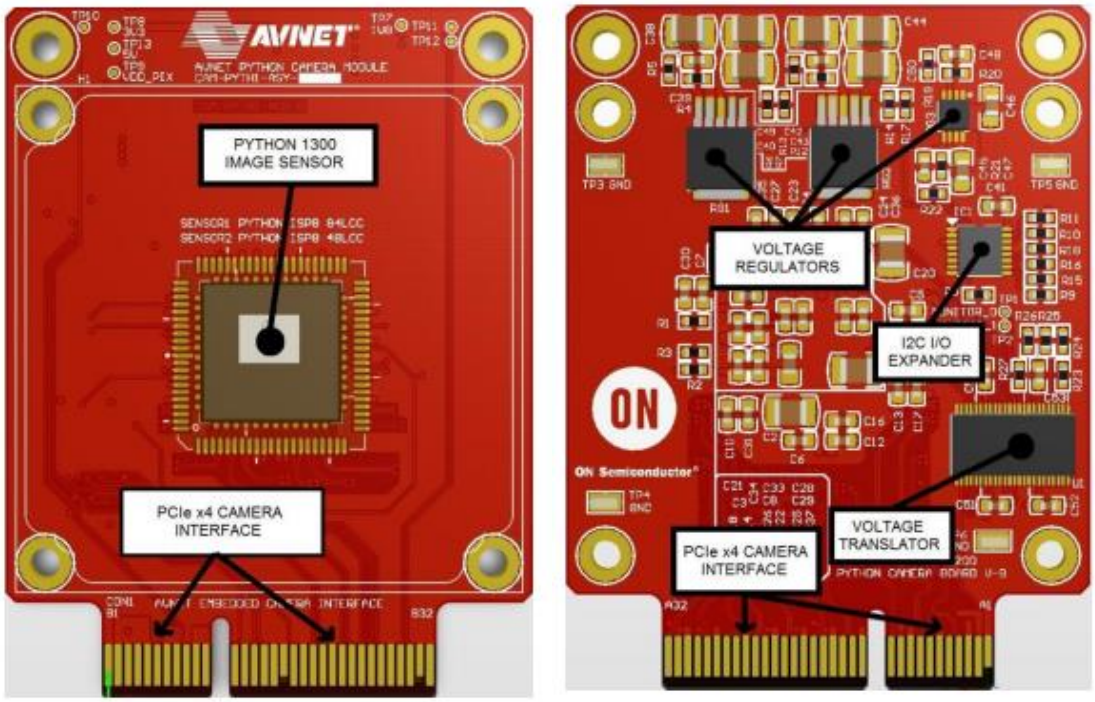


Figura 2.9: Módulo cámara PYTHON 1300-C [6]

En la Figura 2.10 se muestra un diagrama más detallado de la tarjeta:

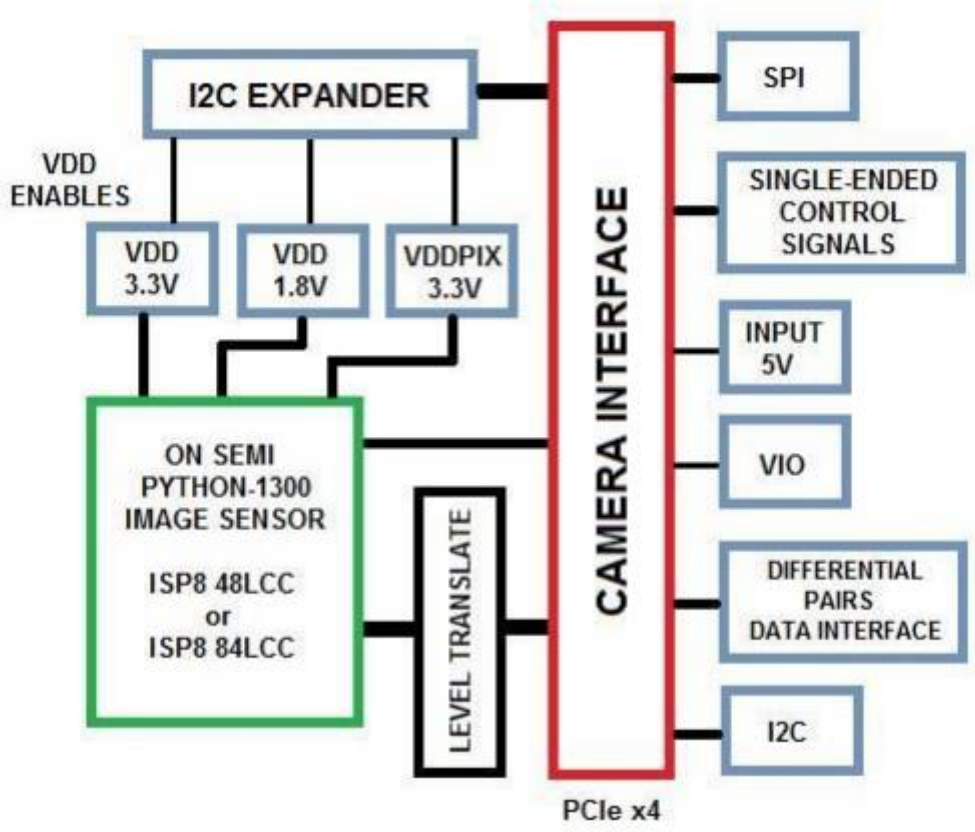


Figura 2.10: Diagrama de bloques del módulo de cámara [6]

La interfaz I2C que provee la tarjeta EMBV está conectada a un expansor I/O PCA9654 de ON Semiconductor. Este dispositivo permite disponer de 8 pines GPIO que son controlados por el usuario a través del protocolo I2C. En esta tarjeta, únicamente se emplean 3 de estos pines, cada uno de ellos conectados a la señal de habilitación de cada uno de los reguladores de tensión, tal y como se puede apreciar en la tabla que se muestra en la Figura 2.11.

Control Signal	Function	Default Value	Direction	Expander Port
ENABLE_VDD_18	1.8V Power Regulator Enable	Pulled-Low	Output	P0
ENABLE_VDD_33	3.3V Power Regulator Enable	Pulled-Low	Output	P1
ENABLE_VDD_PIX	3.3V Power Regulator Enable	Pulled-Low	Output	P2
REV0	Revision GPIO	Pulled-Low	Input	P3
REV1	Revision GPIO	Pulled-Low	Input	P4
REV2	Revision GPIO	Pulled-Low	Input	P5
REV3	Revision GPIO	Pulled-Low	Input	P6
REV4	Revision GPIO	Pulled-Low	Input	P7

Figura 2.11: Conexiones Expansor I2C PCA9654 módulo cámara [6]

La inicialización y configuración del sensor PYTHON se realiza empleando el protocolo SPI, tal y como posteriormente se explicará.

El sensor PYTHON utiliza el sistema de señal diferencial de bajo voltaje o LVDS (*Low-Voltage Differential Signaling*) para el envío de los datos capturados por el sensor. Este sistema es uno de los soportados por el dispositivo Zynq por lo que no es necesario introducir ningún dispositivo adicional en la tarjeta.

Capítulo 3: Captura y visualización de imágenes

El objetivo de este capítulo es la puesta en marcha de los diferentes elementos explicados anteriormente, de forma que se capturen imágenes a partir del sensor óptico PYTHON y se saquen por la interfaz HDMI disponible en la tarjeta EMBV. De esta forma, las imágenes se muestran en tiempo real en un monitor conectado a este puerto.

En la Figura 3.1 se puede observar un diagrama donde se muestran los diferentes pasos que se van a ir dando a lo largo de este capítulo hasta lograr el objetivo propuesto.

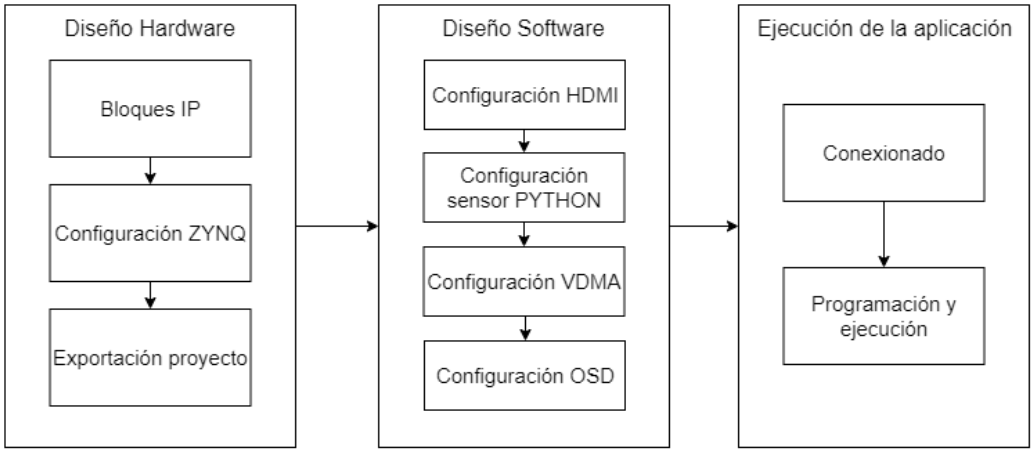


Figura 3.1: Diagrama de flujo del diseño para la captura y visualización de imágenes

3.1 Diseño Hardware

El diseño *hardware* utilizado en este capítulo está basado en la aplicación *embv_p1300c* desarrollada por Avnet y disponible en [7]. En este mismo repositorio se encuentran los 3 bloques IP desarrollados por Avnet que facilitan el uso del sensor PYTHON 1300c y el puerto HDMI de la tarjeta MicroZed.

3.1.1 Implementación en Vivado

Se va a utilizar la herramienta Vivado 2015.4 de Xilinx para realizar el diseño *hardware*. Una vez creado el proyecto, es necesario añadir los 3 bloques IP al proyecto para que posteriormente puedan ser añadidos al diseño. Para ello, es necesario seleccionar la opción *IP Settings* y añadir en la pestaña *Repository Manager* la carpeta *IP* donde se encuentran los bloques *hardware* diseñados por Avnet.

Una vez realizado el paso anterior, los bloques IP de control de la cámara, *ON Semiconductor PYTHON SPI Controller* y *ON Semiconductor PYTHON Camera Receiver*, y el módulo de control del dispositivo HDMI de salida, *Avnet HDMI Output (for ADV7511)*, estarán disponibles en la lista de *IP Cores* para ser añadidos al proyecto.

3.1.2 Bloques HW

El diseño *hardware* llevado a cabo se puede observar en la Figura 3.2:

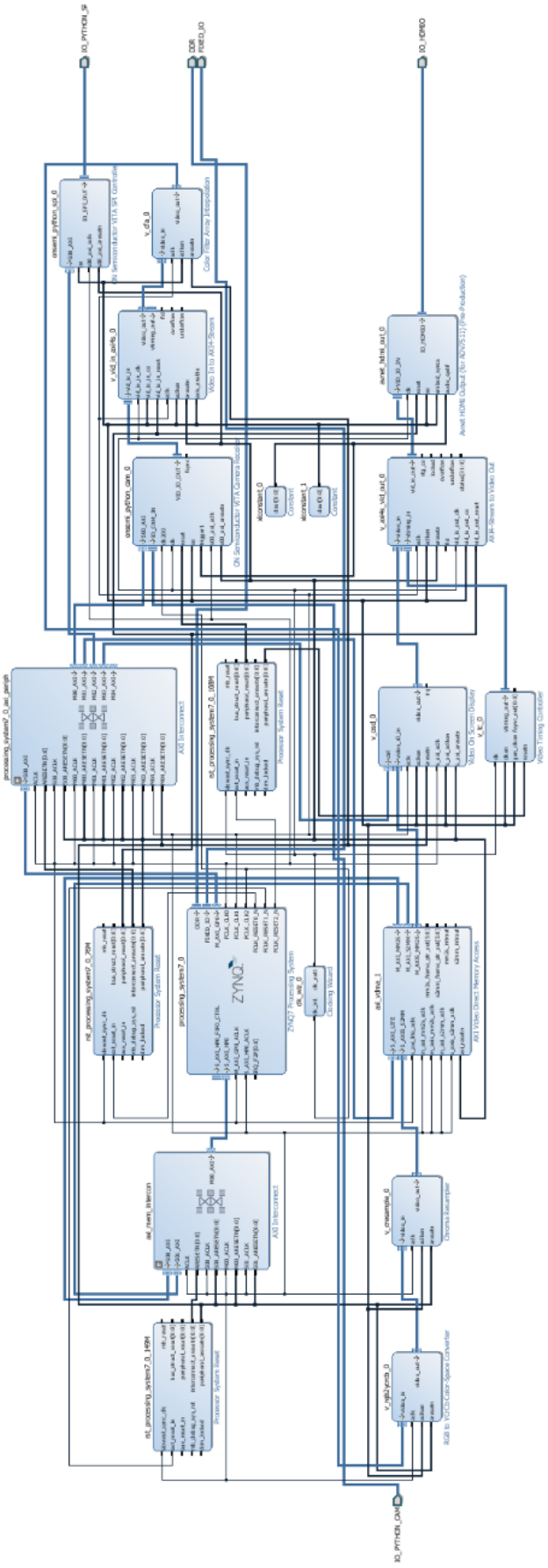


Figura 3.2: Diseño bloques Vivado

A continuación se enumeran los bloques más importantes utilizados en este diseño. Algunos de estos IP Cores desarrollados por Xilinx requieren disponer de una licencia específica para su configuración y posterior implementación:

- ON Semiconductor PYTHON SPI Controller
- ON Semiconductor PYTHON Camera Receiver
- Video In to AXI 4-Stream
- Color Filter Array Interpolation (licencia)
- RGB to YCrCb Color-Space Converter
- Chroma Resampler (licencia)
- AXI Video Direct Memory Access
- Video On Screen Display (licencia)
- Video Timing Controller
- AXI 4-Stream to Video Out
- Avnet HDMI Output (for ADV7511)
- ZYNQ7 Processing System

Los diferentes bloques se explican detalladamente en los apartados siguientes.

3.1.2.1 ON Semiconductor PYTHON SPI Controller

Bloque proporcionado por Avnet que se encarga de implementar la comunicación SPI con el sensor óptico. Este protocolo es el implementado en el sensor PYTHON para llevar a cabo la inicialización y configuración de sus registros.

El puerto de salida de este bloque está conectado a un puerto externo que representa las señales SPI físicas del sensor PYTHON. Este puerto debe ser declarado por el usuario, indicando las diferentes características de las señales a las que representa. Para ello, se emplea un archivo de restricciones o constraints XDC.

Las siguientes líneas del archivo *constraints* relacionan a qué pines físicos del dispositivo Zynq están conectadas las señales SPI del sensor óptico. Esta relación de señales y pines, representados por una letra seguida de dos números, se fija utilizando la propiedad *PACKAGE_PIN* y puede ser consultada en [3].

```
set_property PACKAGE_PIN M15 [get_ports IO_PYTHON_SPI_spi_mosi]
set_property PACKAGE_PIN M14 [get_ports IO_PYTHON_SPI_spi_sclk]
set_property PACKAGE_PIN N15 [get_ports IO_PYTHON_SPI_spi_ssel_n]
set_property PACKAGE_PIN N16 [get_ports IO_PYTHON_SPI_spi_miso]
```

Es necesario también reflejar el estándar de la tensión de funcionamiento de los pines que soportan la comunicación SPI:

```
set_property IOSTANDARD LVCMOS25 [get_ports IO_PYTHON_SPI_spi_mosi]
set_property IOSTANDARD LVCMOS25 [get_ports IO_PYTHON_SPI_spi_sclk]
set_property IOSTANDARD LVCMOS25 [get_ports IO_PYTHON_SPI_spi_ssel_n]
set_property IOSTANDARD LVCMOS25 [get_ports IO_PYTHON_SPI_spi_miso]
```

3.1.2.2 ON Semiconductor PYTHON Camera Receiver

Bloque proporcionado por Avnet que se encarga de recibir las imágenes del sensor PYTHON 1300. Estos datos son recibidos a través de un puerto externo que se ha denominado *IO_PYTHON_CAM*.

Al igual que en el apartado anterior, es necesario indicar cuales son los pines físicos del dispositivo Zynq conectados a los pines del sensor óptico PYTHON:

```
set_property PACKAGE_PIN G14 [get_ports {IO_PYTHON_CAM_trigger[0]}]
set_property PACKAGE_PIN J15 [get_ports {IO_PYTHON_CAM_trigger[1]}]
set_property PACKAGE_PIN B20 [get_ports {IO_PYTHON_CAM_trigger[2]}]
set_property PACKAGE_PIN U13 [get_ports IO_PYTHON_CAM_reset_n]
set_property PACKAGE_PIN B19 [get_ports {IO_PYTHON_CAM_monitor[0]}]
set_property PACKAGE_PIN A20 [get_ports {IO_PYTHON_CAM_monitor[1]}]
set_property PACKAGE_PIN C20 [get_ports IO_PYTHON_CAM_clk_pll]
set_property PACKAGE_PIN H16 [get_ports IO_PYTHON_CAM_clk_out_p]
set_property PACKAGE_PIN H17 [get_ports IO_PYTHON_CAM_clk_out_n]
set_property PACKAGE_PIN K17 [get_ports IO_PYTHON_CAM_sync_p]
set_property PACKAGE_PIN K18 [get_ports IO_PYTHON_CAM_sync_n]
set_property PACKAGE_PIN M19 [get_ports {IO_PYTHON_CAM_data_p[0]}]
set_property PACKAGE_PIN M20 [get_ports {IO_PYTHON_CAM_data_n[0]}]
set_property PACKAGE_PIN L19 [get_ports {IO_PYTHON_CAM_data_p[1]}]
set_property PACKAGE_PIN L20 [get_ports {IO_PYTHON_CAM_data_n[1]}]
set_property PACKAGE_PIN F16 [get_ports {IO_PYTHON_CAM_data_p[2]}]
set_property PACKAGE_PIN F17 [get_ports {IO_PYTHON_CAM_data_n[2]}]
set_property PACKAGE_PIN E18 [get_ports {IO_PYTHON_CAM_data_p[3]}]
set_property PACKAGE_PIN E19 [get_ports {IO_PYTHON_CAM_data_n[3]}]
```

Las siguientes líneas indican el voltaje asociado a cada uno de los pines utilizados:

```
set_property IOSTANDARD LVCMOS25 [get_ports
{IO_PYTHON_CAM_trigger[2]}]
set_property IOSTANDARD LVCMOS25 [get_ports
{IO_PYTHON_CAM_trigger[1]}]
set_property IOSTANDARD LVCMOS25 [get_ports
{IO_PYTHON_CAM_trigger[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports IO_PYTHON_CAM_reset_n]
set_property IOSTANDARD LVCMOS25 [get_ports
{IO_PYTHON_CAM_monitor[1]}]
set_property IOSTANDARD LVCMOS25 [get_ports
{IO_PYTHON_CAM_monitor[0]}]
set_property IOSTANDARD LVCMOS25 [get_ports IO_PYTHON_CAM_clk_pll]
set_property IOSTANDARD LVDS_25 [get_ports IO_PYTHON_CAM_clk_out_*]
set_property IOSTANDARD LVDS_25 [get_ports IO_PYTHON_CAM_sync_*]
set_property IOSTANDARD LVDS_25 [get_ports IO_PYTHON_CAM_data_*]
```

Por último, se indica con *DIFF_TERM* seguido por la palabra *true* las señales del puerto que son de tipo diferencial:

```
set_property DIFF_TERM true [get_ports IO_PYTHON_CAM_clk_out_*]
set_property DIFF_TERM true [get_ports IO_PYTHON_CAM_sync_*]
set_property DIFF_TERM true [get_ports IO_PYTHON_CAM_data_*]
```

Este bloque IP emplea dos señales de reloj, una de 200 MHz necesaria para la correcta recepción de los datos enviados por el sensor y otra de 108 MHz que será la frecuencia de salida de cada uno de los píxeles.

3.1.2.3 Video In to AXI 4-Stream

Este bloque convierte los datos de una fuente de vídeo a datos de tipo Vídeo AXI4-Stream para que posteriormente sean procesados.

El protocolo *AXI4-Stream* para la transmisión de vídeo emplea 5 señales:

- *m_axis_video_tvalid*: Determina si está habilitada o no la transmisión de datos de vídeo a través del bus *AXI4-Stream*.

- *m_axis_video_tdata*: Conjunto de señales por las cuales se transmiten en paralelo los datos. El número de señales debe ser múltiplo de 8 bits y, por tanto, si los datos no son enteros múltiplos de 8, estos deben rellenarse con ceros en los bits más significativos.
- *m_axis_video_tuser*: Señal que indica el inicio de cada *frame*.
- *m_axis_video_tlast*: Señal que indica el final de cada línea del *frame*.
- *m_axis_video_tready*: Señal para sincronizar el envío y recepción de datos a través del bus *AXI4-Stream*. Esta señal permite a un esclavo indicar que está preparado para recibir datos y, por tanto, cuando el maestro puede iniciar la transmisión.

Así, a partir de los datos de la fuente de vídeo, el bloque añade la señal SOF (*Start of Frame*) transmitida en la señal *m_axis_video_tuser* y que indica el píxel en el que comienza a transmitirse una imagen nueva y la señal EOL (*End of Line*) transmitida por la señal *m_axis_video_tlast* que indica cual es el último píxel de cada línea que compone un *frame*.

La configuración del bloque IP se muestra en la Figura 3.3:

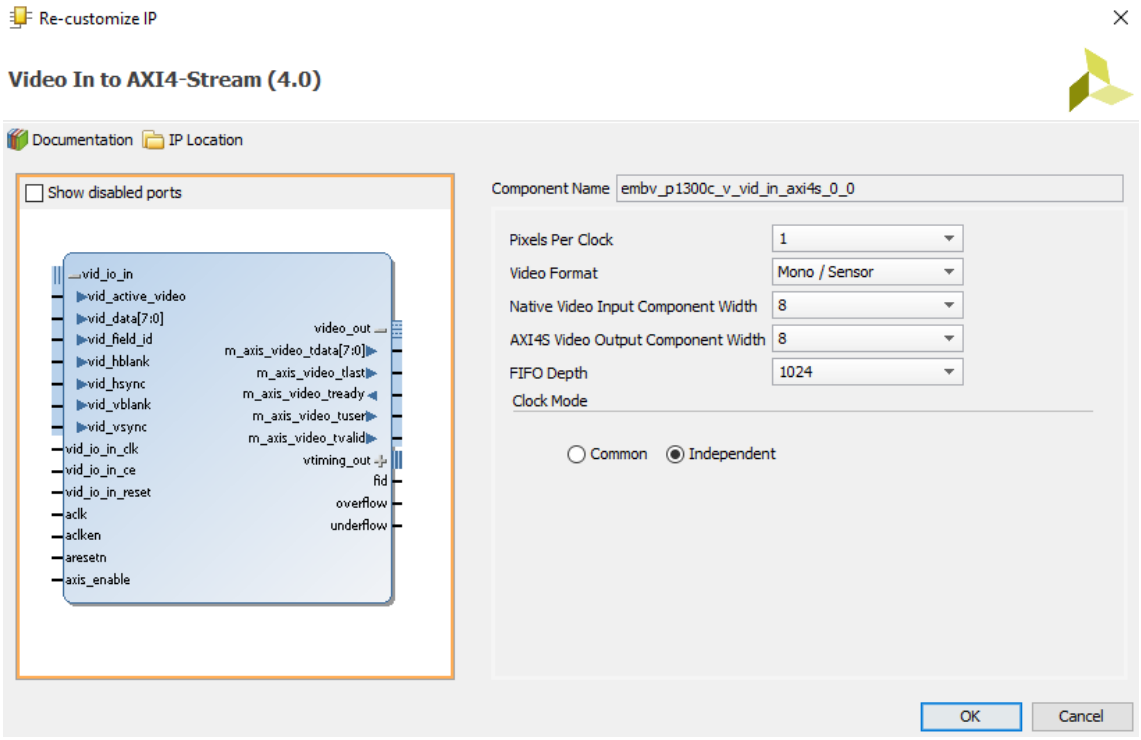


Figura 3.3: Configuración bloque IP Video In to AXI4-Stream

La opción *Pixels Per Clock* permite indicar cuantos píxeles en paralelo se obtienen por ciclo de reloj en el bus *AXI4-Stream* de salida. En este caso, se ha configurado un único píxel por ciclo.

El apartado *Video Format* indica como es el formato que tienen los datos de vídeo a la entrada del bloque. En este caso, se ha seleccionado la opción *Mono/Sensor* que indica que los datos de vídeo de entrada son un único canal y provienen directamente de una cámara. En esta opción se podría configurar otros formatos como *RGB* o *YUV*. En ambos casos, los canales que conforman dichos formatos se recibirían empaquetados en un único conjunto de datos.

La configuración *Native Video Input Component Width* y *AXI4S Video Output Component Width* indican el número de bits por canal tanto a la entrada como a la salida. En este caso, se han configurado a 8 bits ambas opciones.

La opción *Clock Mode* permite determinar si la señal de entrada y la de salida van a usar el mismo reloj. En el caso de seleccionar la opción *Common*, la entrada a*clk* servirá como reloj tanto para la lectura de los datos de entrada como para la generación de los datos de salida. Sin embargo, la opción *Independent* genera una nueva entrada, denominada *vid_io_in_clk*, que permite indicar la frecuencia a la que llegan nuevos datos de vídeo. En este caso se ha seleccionado la opción *Independent*.

Por último, el apartado *FIFO Depth* indica el tamaño de la FIFO asíncrona de entrada. En este buffer, si se ha seleccionado la opción *Independent*, se escriben los datos de entrada a la frecuencia *vid_io_in_clk* y se leen a la frecuencia del bus *AXI4-Stream*, *aclk*. Esta configuración solo es relevante en el caso de que la frecuencia de los datos de entrada sea mayor a la frecuencia del bus *AXI4-Stream*. En estos casos, es necesario asegurarse de que el buffer no se llene completamente, produciéndose una pérdida de datos al ser estos sobrescritos.

En este diseño, la frecuencia del bus *AXI4-Stream* es de 150 MHz y los datos a la entrada llegan con una frecuencia de 108 MHz. De esta forma, como la frecuencia de reloj del bus *AXI4-Stream* es mayor que la frecuencia de llegada de los datos de vídeo, estos son leídos del buffer antes de que puedan acumularse y, por tanto, el tamaño del buffer puede ser el mínimo posible, 32.

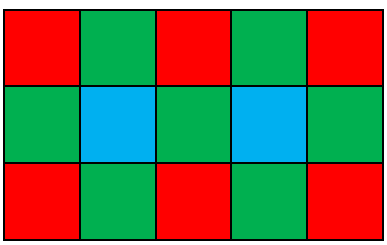
3.1.2.4 Color Filter Array Interpolation

La información de la imagen recibida por el bus *AXI4-Stream* proviene de un sensor CMOS que aplica un filtro de Bayer a la imagen que captura, con el objetivo de conseguir una imagen en color. Para ello, el filtro de Bayer permite el paso de una determinada longitud de onda, eliminando el resto. De esta forma, cada píxel sobre el sensor únicamente da información de uno de los colores primarios: rojo, verde o azul. Así, cada píxel recibido desde el sensor únicamente tiene un canal que da información sobre uno de los colores primarios.

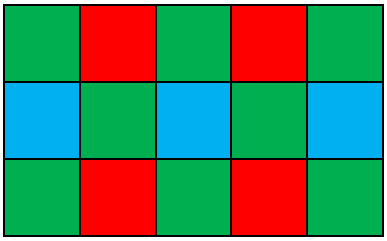
Para obtener variedad de información de los diferentes colores, el filtrado de los píxeles se reparte por toda la imagen según un modelo fijo. Del total de píxeles que componen una imagen, el 50% de los filtros son para el color verde, el 25% para el rojo y el 25% restante para el azul. Se emplean más filtros verdes que de otros colores debido a que el ojo humano es más sensible a este color.

Existen diferentes patrones para repartir el filtrado por la imagen, de forma que, para recuperar esta, es necesario conocer cuál es el que se ha aplicado en un determinado sensor. Los patrones que permite este bloque son los siguientes:

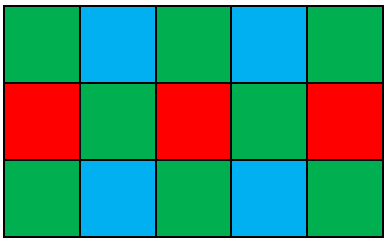
- RGRG



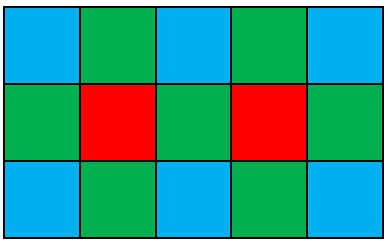
- GRGR



- GBGB



- BGBG



Sin embargo, para la reconstrucción una imagen en color, es necesario que cada píxel tenga información sobre los tres colores primarios y, por tanto, tenga tres canales, uno para cada uno de ellos. Esta información no puede ser recuperada de ninguna forma por lo que se interpola a partir de los píxeles vecinos.

Este bloque tiene como función principal la interpolación de cada píxel generado por el sensor CMOS para conseguir que cada uno de los píxeles que componen la imagen este conformado por 3 canales, uno para cada color. De esta forma, cada píxel tendrá una composición del color en formato RGB.

Este bloque IP se ha configurado según se aprecia en la Figura 3.4:

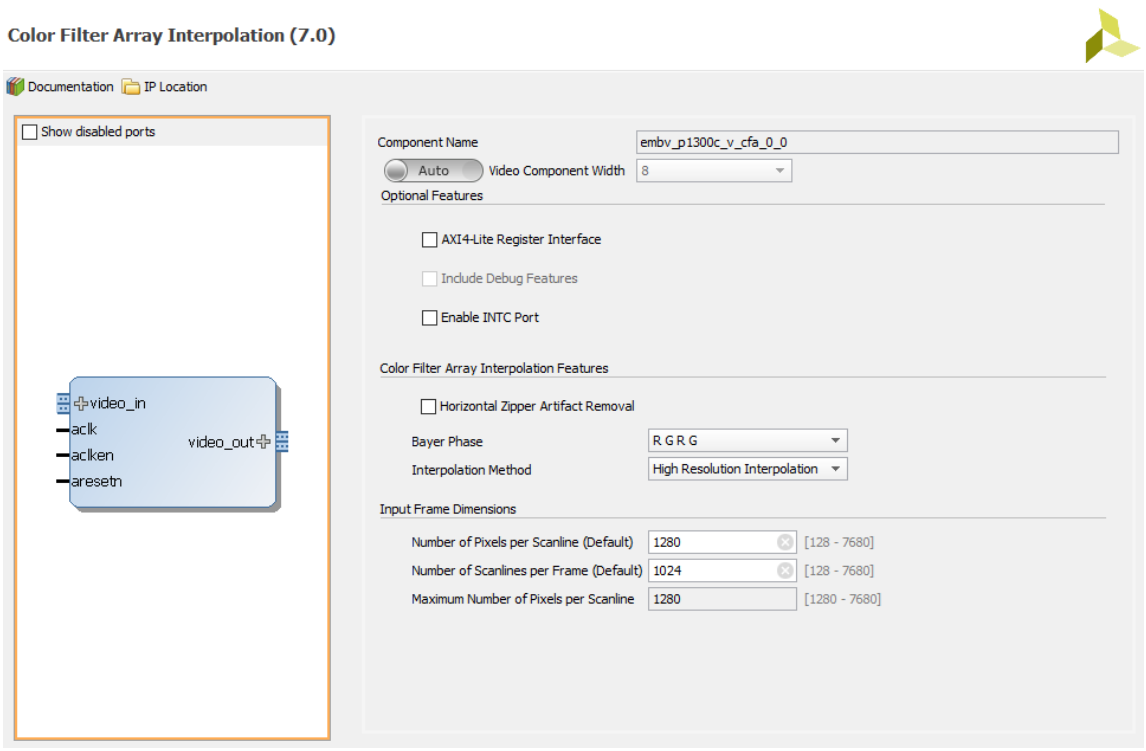


Figura 3.4: Configuración bloque IP Color Filter Array Interpolation

El apartado *Video Component Width* indica el número de bits que tiene cada una de las muestras de entrada. Únicamente puede tomar los valores 8, 10 o 12 bits. Este valor puede ser obtenido automáticamente según el tamaño de los datos del bus *AXI4-Stream* de entrada.

Dentro del apartado *Optional Features*, se pueden seleccionar las siguientes características:

- *AXI4-Lite Register Interface*: Permite la modificación desde el código *software* de algunas de las características de configuración de este bloque. Cuando se habilita esta opción, se añade al bloque una interfaz *AXI4-Lite* a través de la cual posteriormente se llevará a cabo la configuración deseada.
- *Include Debug Features*: Esta opción permite el empleo de características de depuración.
- *Enable INTC Port*: Genera el puerto de salida INTC_IF que da información en paralelo de diferentes fuentes de interrupción: inicio y fin del procesamiento del *frame* e información de la señal de final de línea y de inicio de *frame*.

El siguiente conjunto de apartados de configuración, *Color Filter Array Interpolation Features*, se indican las características necesarias para llevar a cabo el proceso de interpolación:

- *Horizontal Zipper Artifact Removal*: Esta acción agrega a la salida un filtro de suavizado para eliminar defectos denominados *Zipper* o cremallera. Se producen por cambios abruptos en píxeles vecinos y suelen aparecer en las regiones alrededor de los bordes.
- *Bayer Phase*: Esta configuración permite indicar como es el filtro de Bayer aplicado en el sensor. Esta información es necesaria para poder conocer en cada punto de la imagen, que filtrado se ha aplicado en los píxeles vecinos, pudiendo interpolar el valor de los dos canales faltantes para cada píxel. Las opciones disponibles son:

RGRG, GRGR, GBGB y BGBG. En el caso que se ha desarrollado, el sensor Python1300-C aplica un filtro Bayer de tipo RGRG [8].

- *Interpolation Method*: Selecciona el método de interpolación a utilizar. Existen dos opciones disponibles: *Fringe Tolerant Interpolation* y *High Resolution Interpolation*. El primero de los métodos emplea un menor uso de recursos, pero las imágenes de salida tienen peor calidad que las obtenidas con el segundo de los métodos posibles.

El último apartado de la configuración, *Input Frame Dimensions*, permite configurar el tamaño de la imagen de entrada:

- *Number of Active Pixels per Scan line*: indica el número de píxeles por fila que tiene la imagen de entrada, es decir, el número de columnas que tiene cada *frame*. En este caso, el número de píxeles por línea será 1280 ya que esta es la resolución horizontal del sensor de imagen utilizado.
- *Number of Active Lines per Frame*: indica el número de filas que tiene la imagen de entrada. En este caso, este apartado tomará el valor 1024 ya que este es el número de filas que componen una imagen capturada por el sensor PYTHON 1300-C.
- *Maximum Number of Active Pixels Per Scan line*: esta configuración se habilita únicamente cuando la opción *AXI4-Lite Register Interface* está habilitada. Este valor se emplea para conocer el tamaño máximo que deben tener los buffers internos que se emplean para realizar la interpolación. En el caso de que la configuración por *software* esté deshabilitada, el tamaño de los *buffers* es fijo y de tamaño *Number of Active Pixels per Scan line*. Sin embargo, si el valor de columnas de la imagen de entrada puede configurarse por *software*, es necesario indicar cuál será el máximo número de columnas que puede llegar a utilizarse para reservar *buffers* de dicho tamaño.

3.1.2.5 RGB to YCrCb Color-Space Converter

En este punto, la imagen de entrada está conformada por píxeles con tres canales, RGB. En cada uno de los canales se guarda información sobre el nivel de color rojo, verde y azul que, en conjunto, conforman el color que toma el píxel. El procesamiento de imágenes en el espacio de color RGB no es el método más eficiente [9] ya que para modificar la intensidad o el color de un determinado píxel es necesario modificar las tres componentes RGB.

Este bloque permite cambiar el espacio de color desde RGB a YCrCb. El espacio de color YCrCb fue desarrollado como parte de la ITU-R BT.601 y da información sobre la luminancia o luma a través del canal Y, además de sobre la crominancia en los canales Cr y Cb.

La luma da información sobre la cantidad de color blanco que tiene un píxel. Una imagen conformada únicamente por el canal Y es una copia en escala de grises de la imagen a color.

Las componentes de crominancia son las encargadas de transportar información sobre el color de un determinado píxel. Cb y Cr son denominadas diferencia de azul y diferencia de rojo respectivamente.

La configuración que se ha realizado sobre este bloque se puede observar en la Figura 3.5.

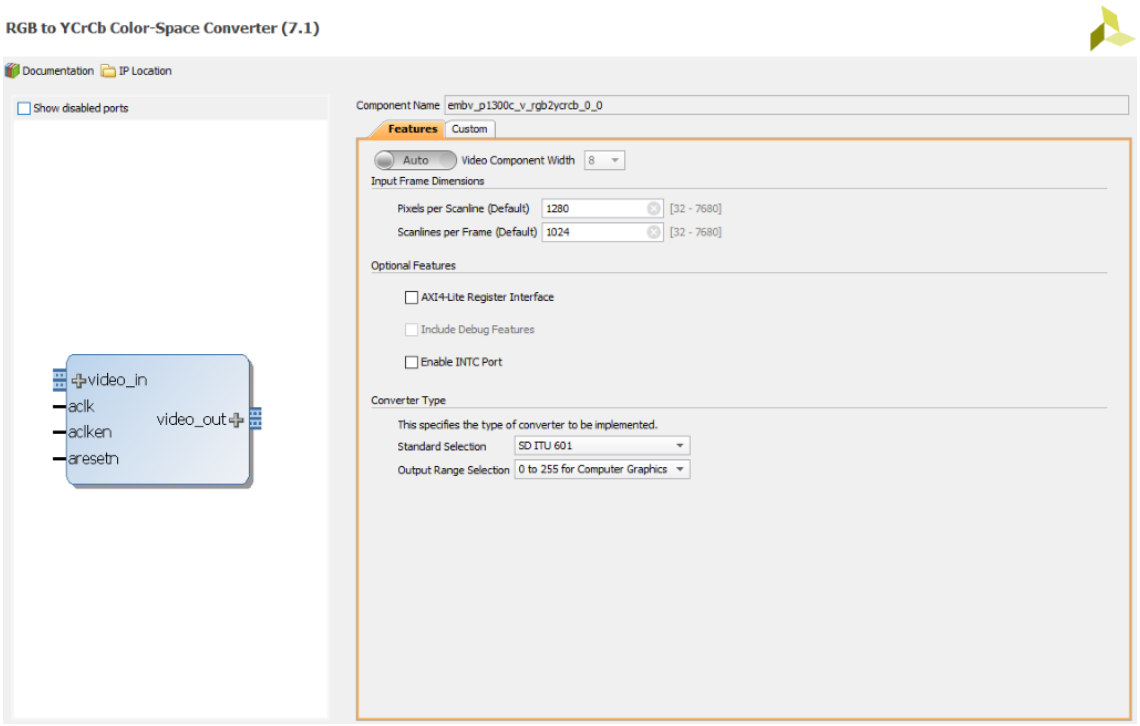


Figura 3.5: Configuración bloque IP RGB to YCrCb Color-Space Converter

El apartado *Video Component Width* permite configurar el tamaño de los datos que componen cada canal de cada píxel y puede tomar los valores 8, 10, 12 o 16 bits. Este parámetro se configura automáticamente según el tamaño de la señal de datos del bus *AXI4-Stream* de entrada.

El siguiente conjunto de configuraciones se agrupan bajo el nombre *Input Frame Dimensions* y permiten configurar el tamaño de la imagen de entrada. El número de píxeles por fila se indica en el apartado *Pixel per Scanline* y el número de filas en *Scanlines per Frame*.

En el apartado *Output Features* se incluyen las mismas configuraciones que en el bloque anterior: *AXI4-Lite Register Interface*, *Include Debug Features* y *Enable INTC Port*.

El último apartado de configuración, *Converter Type* permite seleccionar como se va a llevar a cabo la conversión. El estándar de conversión deseado se selecciona en *Standard Selection* y puede ser uno de los siguientes: *YCrCb ITU 601 (SD)*, *YCrCb ITU 709 (HD) 1125/60 (PAL)*, *YCrCb ITU 709 (HD) 1250/50 (NTSC)* o *YUV*. En estos casos, los parámetros que se emplean para realizar la conversión se pueden observar en la pestaña *Custom*. Estos parámetros pueden elegirse en su totalidad si se selecciona la opción *Custom* en la pestaña *Standard Selection*.

Por último, en el apartado *Output Range Selection* se puede seleccionar el rango de los canales Y, Cr y Cb. Si se desea que el rango de salida sea el máximo posible, se debe seleccionar la opción *0 to 255, typical for computer graphics*.

3.1.2.6 Chroma Resampler

Con el objetivo de reducir el número de datos que son transmitidos se pretende comprimir el flujo de datos. Para ello, aprovechando que el sistema visual humano es más sensible a los cambios de la componente de luminancia que a los cambios de las componentes de crominancia o color, se realiza un submuestreo de crominancia. De esta forma, los datos relativos al color se muestrean a menor frecuencia que los de luminancia.

La configuración de este bloque IP se muestra en la Figura 3.6.

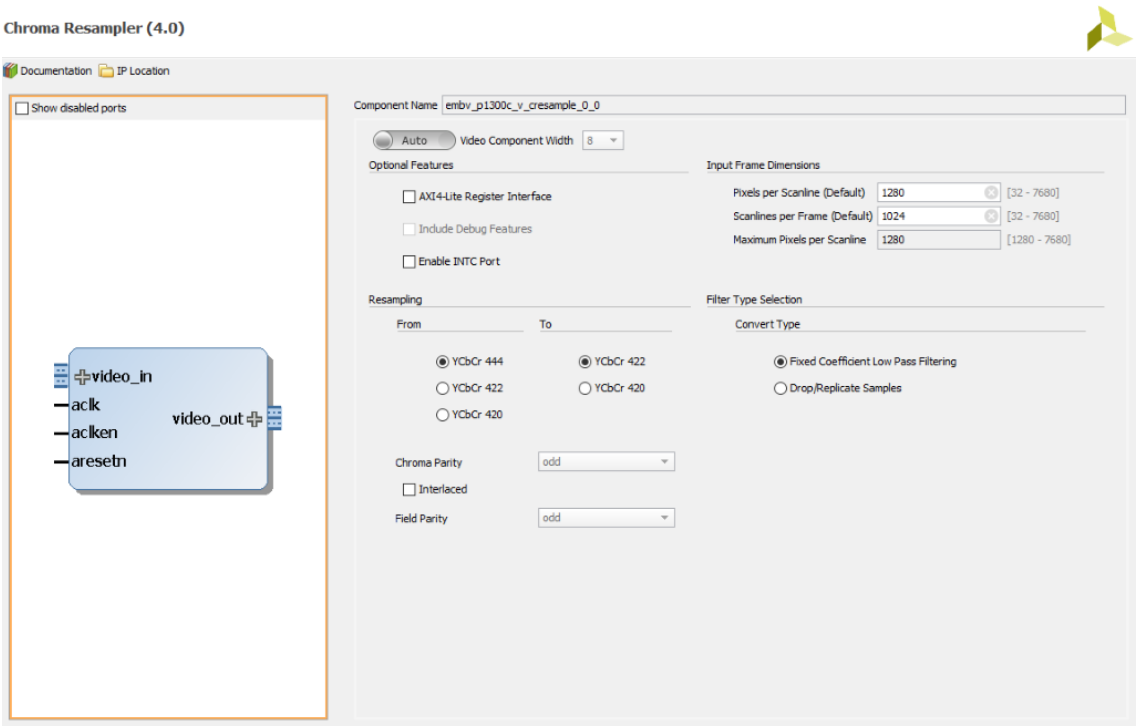


Figura 3.6: Configuración bloque IP Chroma Resampler

El primer parámetro a configurar es el número de bits usados para canal en los datos de entrada. Esta configuración puede automáticamente seleccionar este valor a partir de los datos del bus *AXI4-Stream* de entrada.

Los bloques de configuración *Optional Features* e *Input Frame Dimensions* se emplean para habilitar la configuración del bloque por *software* y para seleccionar el tamaño de la imagen de entrada. Una explicación más detallada de cada configuración se puede encontrar en apartados anteriores.

En el apartado *Resampling* se lleva a cabo toda la configuración necesaria para llevar a cabo el submuestreo de la imagen:

- En el subapartado *From* se selecciona el formato actual de los datos de entrada. Este puede ser:
  - *YCbCr 444*: Este esquema no tiene ningún submuestreo de los componentes de color.

	Columna 0	Columna 1	Columna 2	Columna 3
Fila 0	Y Cb Cr	Y Cb Cr	Y Cb Cr	Y Cb Cr
Fila 1	Y Cb Cr	Y Cb Cr	Y Cb Cr	Y Cb Cr
Fila 2	Y Cb Cr	Y Cb Cr	Y Cb Cr	Y Cb Cr
Fila 3	Y Cb Cr	Y Cb Cr	Y Cb Cr	Y Cb Cr

  - *YCbCr 422*: En este esquema las componenetes Cb y Cr son muestreadas por un factor horizontal 2, es decir, solo en una de cada dos columnas se tienen en cuenta los canales de crominancia. Esto supone que la resolución horizontal de crominancia es la mitad respecto a la luma y la resolución vertical es completa.

	Columna 0	Columna 1	Columna 2	Columna 3
Fila 0	Y Cb Cr	Y	Y Cb Cr	Y
Fila 1	Y Cb Cr	Y	Y Cb Cr	Y
Fila 2	Y Cb Cr	Y	Y Cb Cr	Y
Fila 3	Y Cb Cr	Y	Y Cb Cr	Y

- *YCbCr 420*: Las componentes Cb y Cr son muestreadas por un factor horizontal 2 y un factor vertical 2. En este caso, ambas resoluciones son la mitad de la resolución de luma. El cálculo de la crominancia se lleva a cabo mediante la interpolación de los canales Cb y Cr de ambas filas.

	Columna 0		Columna 1	Columna 2		Columna 3
Fila 0	Y	Cb	Y	Y	Cb	Y
Fila 1	Y	Cr	Y	Y	Cr	Y
Fila 2	Y	Cb	Y	Y	Cb	Y
Fila 3	Y	Cr	Y	Y	Cr	Y

- El subapartado *To* indica cual es el formato que se desea a la salida: *YCbCr 422*, *YCbCr 420* o *YCbCr 444*.
- En el caso de seleccionar el formato *YCbCr 420*, se habilita la opción *Chroma Parity*. Esta opción sirve para indicar si la primera línea de *frame* incluye la información de color (*odd*) o no (*even*).
- La opción *Interlaced* indica si la fuente de vídeo está entrelazada, de forma que se envían primero las líneas pares y después las impares o al revés, o si el vídeo es progresivo y, por tanto, se envían todas las líneas de forma secuencial.
- En el caso de que la fuente esté entrelazada, se debe indicar si primero se envían las líneas pares o las impares.

La generación de los valores de crominancia por interpolación se realiza con un filtro FIR. Los coeficientes pueden ser fijos *Fixed Coefficient Low Pass Filtering* o fijados por el usuario mediante el bus *AXI4-Lite*.

La opción más sencilla *Drop Samples* elimina las muestras de Cb y Cr de un determinado píxel y asume que estas son iguales a los de los píxeles vecinos. En el caso de querer obtener un formato con más muestras desde uno con menos, la opción *Replicate Samples* copia los valores de Cb y Cr de los píxeles cercanos.

A partir de la configuración de este proyecto, los datos de entrada son YCbCr 4:4:4 y se reciben tal y como se puede observar en la Figura 3.7. Cada píxel es transmitido en datos de 24 bits, 8 bits por cada canal.

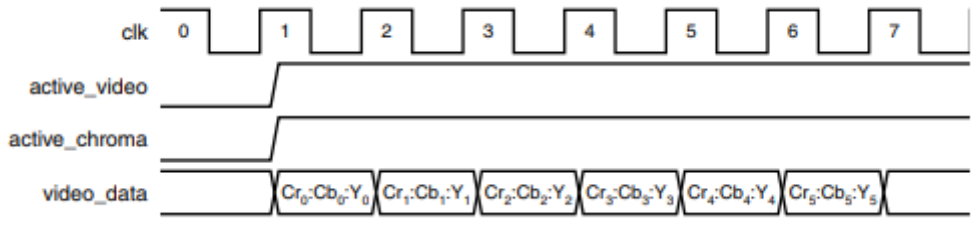


Figura 3.7: Datos entrada en formato YCbCr 4:4:4 [10]

Una vez que los datos se han remuestreado siguiendo un formato YCbCr 4:2:2, cada píxel ocupa 16 bits y los datos se envían según se muestra en la Figura 3.8. Se puede observar que en el primer ciclo se envía el canal de luminancia y el de crominancia de azules. En el segundo ciclo de reloj, se envía un nuevo valor de luma y el dato del canal de crominancia de rojos obtenido del píxel anterior. De esta forma, cada píxel enviado ocupa únicamente 16 bits.

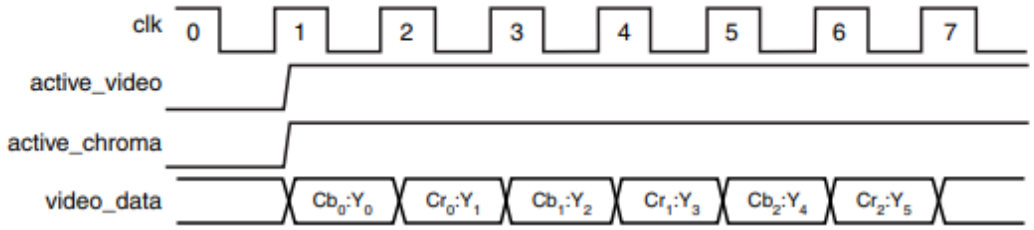


Figura 3.8: Datos salida en formato YCbCr 4:2:2 [10]

3.1.2.7 AXI Video Direct Memory Access - VDMA

A la salida del bloque *Chroma Resampler*, la imagen se conforma con un valor de luminancia en cada ciclo de reloj y, de forma alternada, un valor de crominancia azul o roja. De esta forma, la imagen se comprime de forma que se guardan todos los valores de luminancia y la mitad de los valores de crominancia (formato 4:2:2).

El bloque VDMA se encarga de recibir el flujo de datos *AXI4-Stream* provenientes del bloque anterior y almacenarlos en memoria DDR empleando para ello un puerto AXI4 Maestro y *Direct Memory Access* o DMA.

De la misma forma, el bloque VDMA se encarga de leer, empleando un bus AXI4 Maestro, los datos almacenados en memoria DDR y convertirlos en un flujo de datos para su posterior visualización en un monitor por HDMI.

La principal utilidad de este bloque es su capacidad de aislar los flujos de entrada y de salida de datos. Para ello, emplea una serie de *buffers* en memoria DDR en los cuales realiza las operaciones de escritura y lectura de datos. Para evitar escribir en un *buffer* que está siendo leído o leer en uno que está siendo escrito, se dispone de distintos métodos de sincronización dependiendo las necesidades de una determinada aplicación.

De esta forma, si la generación de datos es más rápida que su consumo, se evita escribir en el *buffer* que está siendo leído, saltando dicho *buffer* y, si es necesario, sobrescribiendo los ya escritos. De la misma forma, si el consumo es más rápido que la generación, es posible repetir los *frames* ya leídos hasta que la escritura de un nuevo *frame* haya terminado. Así, se consigue que el flujo de datos sea continuo, repitiendo o actualizando los *frames* de forma que los datos almacenados sean siempre los más actualizados.

3.1.2.7.1 Canales de escritura y lectura

El bloque VDMA puede ser configurado para actuar únicamente como canal de escritura, únicamente como canal de lectura o como ambos simultáneamente.

3.1.2.7.1.1 Canal de escritura

El canal de escritura permite al bloque capturar un flujo de datos de entrada y almacenarlo en un determinado *buffer* en memoria DDR. La habilitación de este canal activa las siguientes interfaces al bloque:

- *S_AXIS_S2MM*: interfaz esclava del protocolo *AXI4-Stream* a través de la cual se obtienen los datos a escribir en memoria.
- *s_axis_s2mm_aclk*: señal de reloj asociada a la interfaz esclava *AXI4-Stream*.
- *M_AXI_S2MM*: interfaz maestra del protocolo *AXI4*.
- *m_axi_s2mm*: señal de reloj asociada a la interfaz *M_AXI_S2MM*.
- *S2MM_FRAME_PTR_IN[5:0]*: bus de entrada para sincronización.
- *S2MM_FRAME_PTR_OUT[5:0]*: bus de salida para sincronización.

3.1.2.7.1.2 Canal de lectura

El canal de lectura se encarga de leer los datos de un determinado *buffer* en DDR y convertirlos a un flujo de datos. La activación del canal de lectura habilita los siguientes puertos en el bloque VDMA:

- *M_AXIS_MM2S*: interfaz maestra del protocolo *AXI4-Stream* en la cual se escriben los datos leídos de memoria.
- *m_axis_mm2s_aclk*: señal de reloj asociada a la interfaz maestra *AXI4-Stream*.
- *M_AXI_MM2S*: interfaz maestra del protocolo *AXI4*.
- *m_axi_mm2s*: señal de reloj asociada a la interfaz *M_AXI_MM2S*.
- *MM2S_FRAME_PTR_IN[5:0]*: bus de entrada para sincronización.
- *MM2S_FRAME_PTR_OUT[5:0]*: bus de salida para sincronización.

3.1.2.7.2 Sincronización

El bloque VDMA permite dos modos de sincronización directa:

3.1.2.7.2.1 Maestro y esclavo

Con esta configuración, el *master* no monitoriza el estado del *slave*. Así, la señal **_FRAME_PTR_OUT* del maestro está conectada directamente a la señal **_FRAME_PTR_IN* del esclavo, tal y como se puede observar en la Figura 3.9.

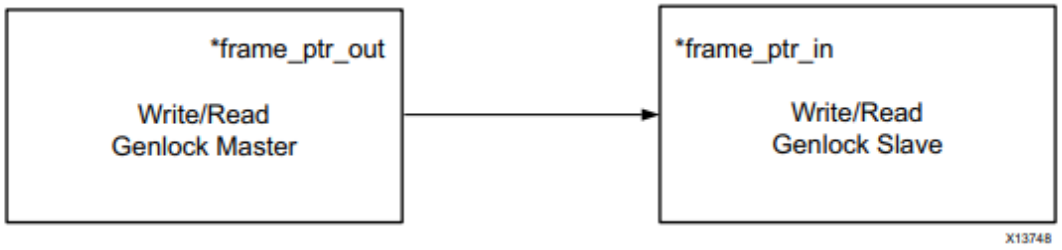


Figura 3.9: Configuración maestro-esclavo [11]

El canal que actúa como maestro, ya sea de escritura o de lectura, no salta ni repite ningún *frame*. El número de *buffer* que se está procesando se indica en el bus *MM2S_FRAME_PTR_OUT* si el canal de lectura actúa como maestro o en el *S2MM_FRAME_PTR_OUT* si es el canal de escritura el que actúa como maestro.

El canal que actúa como esclavo intenta alcanzar al maestro saltando (si es más lento que el maestro) o repitiendo (si es más rápido que el maestro) fotogramas. Así, a partir del *buffer* que el maestro está procesando, conocido por el esclavo a través de la conexión que se muestra en la Figura 3.9, este accede a un *frame* anterior determinado por un valor de retardo configurable. El número de *frame* que se está procesando se indica a través del bus **_FRAME_PTR_OUT*.

3.1.2.7.2.2 Maestro dinámico y esclavo dinámico

En esta configuración, las señales de sincronización de maestro y esclavo están conectadas entre sí, tal y como se puede observar en la Figura 3.10.

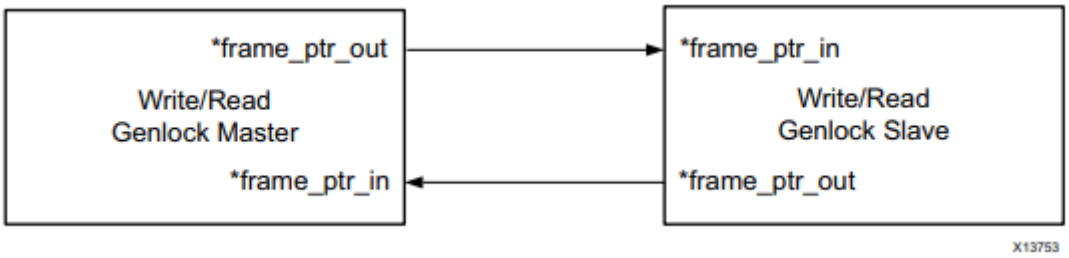


Figura 3.10: Configuración maestro dinámico - esclavo dinámico [11]

El canal que actúa como maestro dinámico evita utilizar el *buffer* que está siendo utilizado por el esclavo dinámico. Para evitar operar sobre un determinado *buffer* el maestro puede utilizar el resto de *frames*, saltando el que está siendo usado por el esclavo dinámico, o repetir un determinado *frame*, sobrescribiendo un mismo *buffer* o leyendo varias veces el mismo *frame*. El bus **_FRAME_PTR_OUT* indica cual ha sido el último *frame* accedido, es decir, cual es el último *buffer* escrito completamente en el caso del canal de escritura o cual es el último *frame* leído en el caso del canal de lectura.

El canal que actúa como esclavo dinámico accede al último *frame* que procesó el maestro dinámico, indicado por el bus **_FRAME_PTR_OUT* del canal maestro. El bus **_FRAME_PTR_OUT* del esclavo refleja cual es el *frame* que actualmente está siendo procesado.

3.1.2.7.2.3 Puertos

Como se ha comentado anteriormente, cada canal habilitado dispone de dos buses de sincronización: **_FRAME_PTR_IN* y **_FRAME_PTR_OUT*. Sin embargo, cuando se habilitan los dos canales (escritura y lectura) y uno de ellos es configurado como maestro y el otro como esclavo o como maestro dinámico y esclavo dinámico, la conexión se realiza de forma interna, no estando disponible los puertos **_FRAME_PTR_IN* accesibles desde el exterior.

Cabe destacar que los buses **_FRAME_PTR_** emplean un código *Gray* para representar cada uno de los *buffers* posibles. Para poder utilizar estos códigos *Gray* de forma circular en un número de *buffers* que no sea potencia de dos, es necesario duplicar estos. Por ejemplo, si se configura el uso de 3 *buffers*, el código *Gray* utilizado será: 1, 3, 2, 6, 7 y 5. Este código se repetirá de forma cíclica. Una tabla con todas las combinaciones posibles según el número de *buffers* utilizados se puede consultar en [11].

3.1.2.7.3 Configuración HW

La configuración de este bloque para esta aplicación se puede observar en la Figura 3.11:

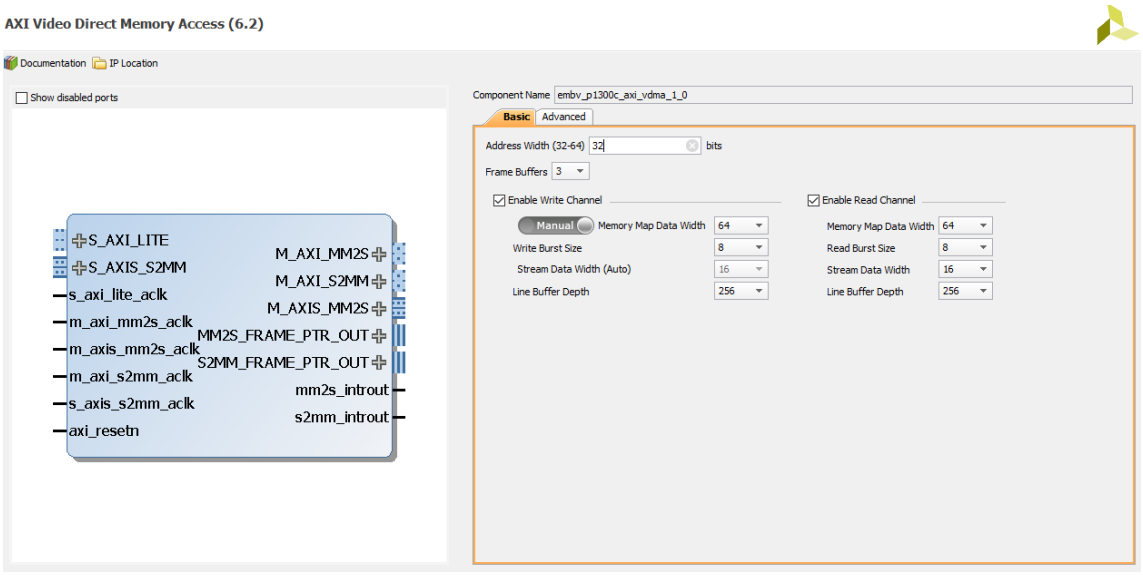


Figura 3.11: Configuración bloque IP VDMA

En el apartado *Basic* de la configuración, se pueden seleccionar los parámetros básicos de funcionamiento de este IP Core.

El campo *Address Width* permite indicar el espacio de direccionamiento pudiendo tomar este cualquier valor entre 32 y 64 bits.

La opción *Frame Buffers* se emplea para seleccionar el número de *buffers* que puede emplear el bloque VDMA para su funcionamiento. En el caso de esta aplicación, se han seleccionado 3 *buffers*.

Dependiendo del uso que se desea dar a este bloque, se puede habilitar o no los canales de escritura y lectura. En este caso, se emplearán ambos canales, el de escritura para almacenar las imágenes capturadas por la cámara y el de lectura para su posterior visualización.

Si se habilita el canal de escritura (*Enable Write Channel*), se habilita un puerto *S_AXIS_S2MM*, a través del cual se recibe el flujo de datos que se desea escribir en memoria y un puerto *M_AXI_S2MM* por el que se realiza el envío de los datos hasta memoria. Estos puertos son un bus *AXI4-Stream* esclavo y un bus AXI4 Máster cuyas características asociadas se deben indicar en los siguientes apartados:

- *Memory Map Data Width:* selecciona el número de bits utilizados para el envío de datos por el canal AXI4 de escritura. Los valores válidos son: 32, 64, 128, 256, 512 y 1024.
- *Write Burst Size:* especifica el máximo número de datos escritos en paralelo. Valores grandes aumentan el rendimiento, pero disminuyen el ancho de banda para otros accesos a memoria en paralelo.
- *Stream Data Width (Auto):* indica la anchura de datos en bits del canal *AXI4-Stream*. Los valores válidos son los múltiplos de 8 hasta 1024. Este valor debe ser menor o igual al indicando en el campo *Memory Map Data Witdh*.
- *Line Buffer Depth:* selecciona el tamaño de cada línea del *buffer*. Se indica en número de veces la anchura del flujo de datos.

De la misma forma, la habilitación del canal de lectura (*Enable Read Channel*) genera un puerto *AXI4-Stream* maestro, *M_AXIS_MM2S*, por el que se envía el flujo de datos proveniente de memoria y un puerto AXI4 maestro, *M_AXI_MM2S*, por el que el bloque

VDMA recibe los datos de memoria. Este canal de lectura tiene los mismos apartados de configuración que el canal de escritura.

En la pestaña *Advanced* se encuentran las opciones de configuración relacionadas con la sincronización de los diferentes canales utilizados. La configuración usada para esta aplicación se puede observar en la Figura 3.12.

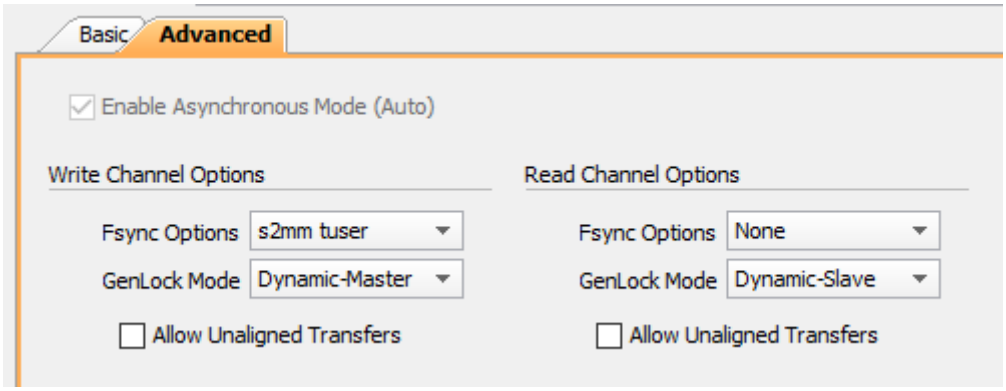


Figura 3.12: Configuración pestaña Advanced VDMA

- *Enable Asynchronous Mode*: permite que las diferentes señales de reloj conectadas a este bloque IP (*m_axi_mm2s_aclk*, *m_axi_s2mm_aclk*, *s_axi_lite_aclk*, *m_axis_mm2s_aclk* y *s_axis_s2mm_aclk*) sean distintas entre sí. Si esta opción está deshabilitada, todas las señales de reloj deben ser la misma.
- Cada canal de escritura o lectura tiene las siguientes opciones:
    - *Fsync Options*: esta opción permite la habilitación de distintos modos de sincronización entre *frames*:
        - *None*: Con esta opción, el bloque VDMA transfiere tan rápido como es posible sin esperar a ninguna señal de disparo o sincronización.
        - *s2mm fsync*: Cuando esta opción esta seleccionada, el bloque VDMA empieza el procesamiento de cada *frame* en cada flanco de bajada de la señal *s2mm_fsync* para el canal de escritura y de la señal *mm2s_fsync* para el canal de lectura.
        - *s2mm tuser*: únicamente disponible para el canal de escritura. Cuando se selecciona esta opción, el bloque VDMA espera la llegada del inicio de *frame* (*start of frame*, SOF) a través de la señal *s_axis_s2mm_tuser(0)* del bus *AXI4-Stream*. Esta señal SOF es un pulso que coincide con el primer píxel enviado de cada *frame*.
    - *Genlock Mode*: Hay 4 opciones:
        - *Master*: cuando se selecciona, el número de *frame* actualmente en el bus se indica en *s2mm_frame_ptr_out* en el caso del canal de escritura y en *mm2s_frame_ptr_out* para el caso del canal de lectura.
        - *Slave*: cuando esta opción es elegida, el esclavo sigue al maestro con un retardo de los *frames* indicado en *Frame Delay* (configurado posteriormente en *software*). Esto permite saltar o repetir *frames*.
        - *Dynamic Master*: esta configuración permite saltarse los datos almacenados en un determinado *buffer* durante el tiempo que el *Dynamic Slave* esté trabajando en él.
        - *Dynamic Slave*: cuando se selecciona, el bloque VDMA sigue al *Dynamic Master* saltando o repitiendo *frames*.

- *Allow Unaligned Transfers*: Cuando esta opción está seleccionada, se permite el envío de datos o lectura de cualquier byte de datos en memoria. Si esta opción está deseleccionada, la dirección de inicio debe estar alineada con múltiplos de la anchura del canal de datos. De la misma forma, los valores *Horizontal Size* y *Stride* deben ser también múltiplos de dicho valor. El significado de estos términos será explicado más detalladamente en apartados posteriores.

A lo largo de este proyecto se emplea el modo *Dynamic-Master* para el canal de escritura en memoria y el modo *Dynamic-Slave* para el canal de lectura. De esta forma, un nuevo *frame* capturado por la cámara no será escrito en el mismo *buffer* que esté siendo leído para ser mostrado por el monitor, siendo este almacenado en cualquiera de los otros *buffers* disponibles. Además, una vez que se haya terminado de leer un determinado *buffer*, el siguiente *frame* que se leerá será el último escrito completamente en memoria.

3.1.2.8 Video On Screen Display

Este bloque IP permite generar un único flujo de vídeo de salida a partir de la combinación de múltiples fuentes de video de entrada. De esta forma, permite posicionar y combinar los diferentes flujos tal y como se desee para generar el flujo de vídeo a la salida.

Además, este bloque facilita la inclusión de cajas y textos en el flujo de salida, permitiendo formatear estos con relativa libertad: tamaño, color, grosor, entre otras. Para ello, dispone de diferentes controladores gráficos internos que, una vez configurados a nivel *hardware*, son programados desde *software* a través de un puerto *AXI4-Lite*.

En el caso de la aplicación que se está desarrollando, este bloque recibe el flujo de datos generado por el canal de lectura del bloque VDMA y genera un flujo de salida igual a este. De esta forma, este bloque no es estrictamente necesario para esta aplicación, pero será utilizado en posteriores capítulos de este documento.

La configuración del bloque IP para esta aplicación se puede observar en la Figura 3.13 y se explica a continuación.

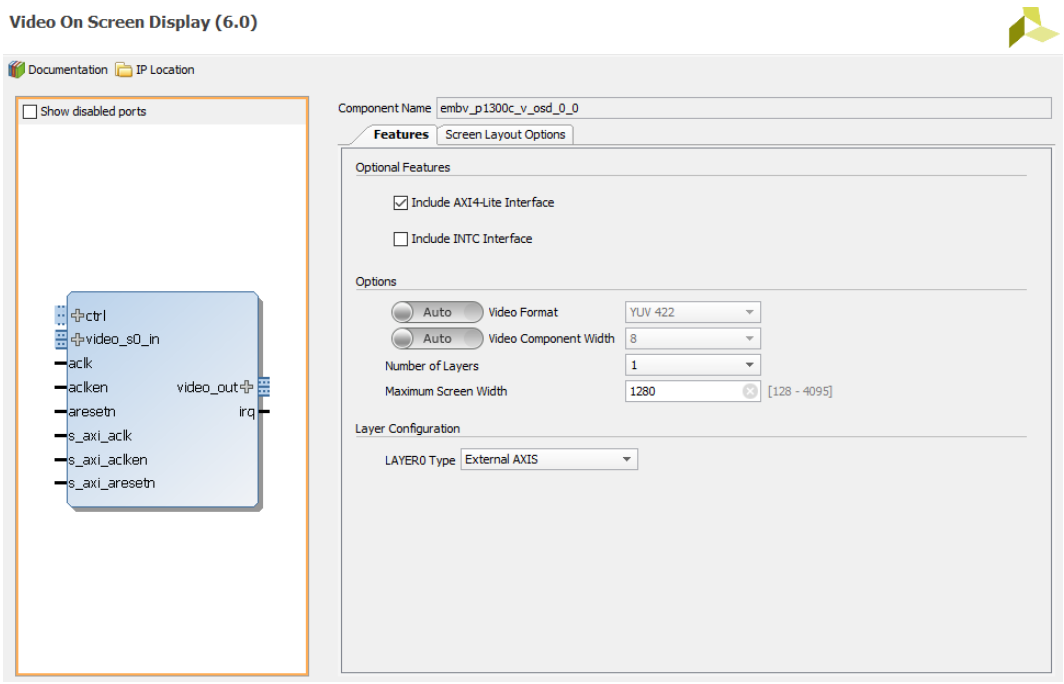


Figura 3.13: Configuración bloque IP Video On Screen Display

La opción *Include AXI4-Lite Interface* permite la programación dinámica y el cambio de parámetros de procesamiento desde el procesador.

En el apartado *Options*, se indican las principales características que configuran el bloque:

- *Video Format*: Este campo indica cual es el formato del flujo de vídeo a la entrada y a la salida. Los formatos válidos son: YUV 422, YUV 444, RGB, YUVa 422 y RGBa. Este parámetro es seleccionado automáticamente.
- *Video Component Width*: Indica el número de bits para cada componente de color. Los valores válidos son: 8, 10 y 12. Este parámetro puede ser seleccionado automáticamente.
- *Number of Layers*: Número de capas a ser combinadas. Este parámetro incluye los flujos de datos de entrada y el número de controladores gráficos internos del bloque. El número máximo de capas es 8. En este caso, únicamente se utiliza una capa que será el flujo de datos de entrada proveniente del bloque VDMA.
- *Maximum Screen Width*: Configura el número máximo de columnas que tendrá la imagen de salida, en este caso, 1280 píxeles.

El apartado *Layer Configuration* específica para cada una de las capas utilizadas si la fuente de datos es externa o si los datos son generados en un controlador gráfico interno:

- *External AXIS*: Genera una interfaz *AXI4-Stream* esclava por la cual se recibirá el flujo de vídeo.
- *Internal Graphics Controller*: Se utiliza el controlador gráfico interno al módulo para la generación de recuadros o textos.

La pestaña *Screen Layout Parameters*, mostrada en la Figura 3.14, se emplea para establecer la composición, a partir de los distintos flujos de entrada, del flujo de salida.

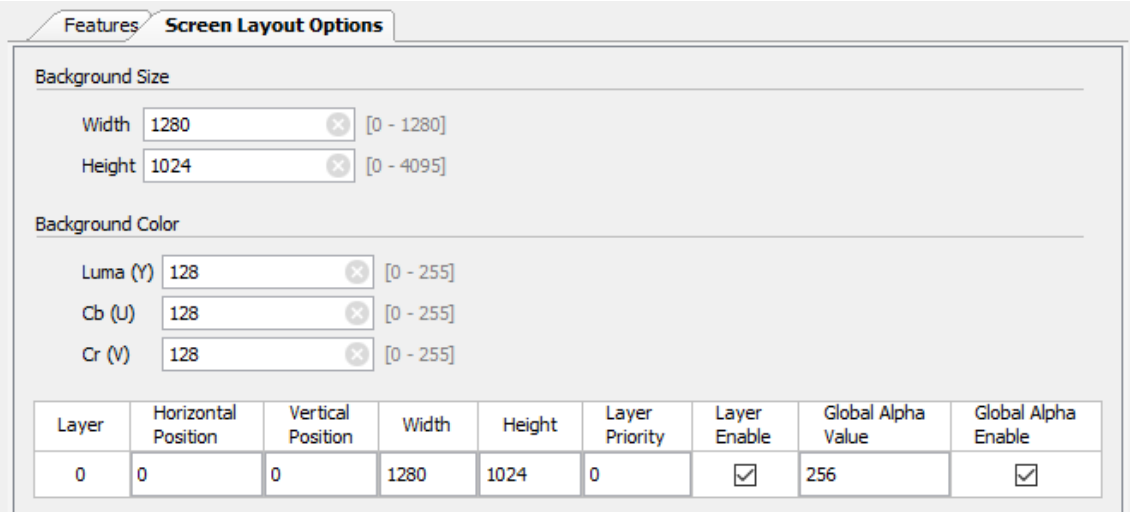


Figura 3.14: Pestaña Screen Layout Parameters del bloque IP OSD

- *Background Size*: indica el tamaño en filas (*height*) y columnas (*width*) del flujo de video a la salida.
- *Background Color*: Color de fondo del flujo de salida. Se indica a partir del valor de los 3 canales que componen cada píxel.
- Para cada capa configurada en la pestaña anterior, se deben completar los siguientes parámetros:

- *Horizontal/Vertical Position*: Configuración de la posición de cada flujo de entrada. La esquina superior izquierda representa el punto 0,0.
- *Width/Height*: Tamaño de la capa.
- *Layer Priority*: Se emplea para determinar que capa se muestra por encima de otras cuando estas se superponen y, por tanto, coinciden en un mismo punto de la imagen de salida. Las capas con alta prioridad se mostrarán por encima de las capas con prioridades inferiores. Cada capa debe tener un único valor de prioridad.
- *Layer Enable*: Configura si la capa está habilitada o deshabilitada por defecto.
- *Global Alpha Value*: Configura el valor *Alpha* de la capa. El valor *Alpha* es utilizado para determinar el grado de transparencia de una determinada capa. Cuanto mayor sea el valor de *Alpha* menor transparencia tendrá dicha capa. El rango de este valor es de 0 a 256.
- *Global Alpha Enable*: Habilita o deshabilita el empleo del valor global de *Alpha* para una determinada capa.

Para cada una de las capas configuradas como *Internal Graphics Controller*, no utilizadas en esta aplicación, se habilita una nueva pestaña, mostrada en la Figura 3.15, que permite configurar el comportamiento de dicha capa.

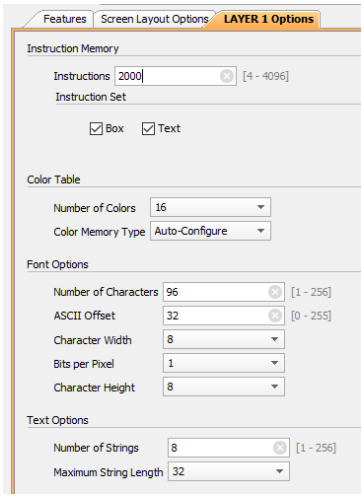


Figura 3.15: Pestaña Internal Graphics Controller del bloque IP OSD

- *Instructions*: Configura el máximo número de instrucciones por *frame* que un determinado controlador gráfico puede ejecutar.
- *Instruction Set*: Indica que tipo de instrucciones son válidas: cajas y/o texto.
- *Number of Colors*: Especifica el tamaño de la paleta de color a utilizar. Las opciones son: 16 o 256.
- *Color Memory Type*: Configura como se implementa en *hardware* la paleta de colores: *Distributed RAM*, *Block RAM* o *Auto-Configured*.
- *Number of Characters*: Indica el número de caracteres que pueden ser almacenados. El rango disponible es 1 a 256.
- *Character Width*: Configura la anchura de cada carácter medido en píxeles.
- *Character Height*: Especifica la altura de cada carácter medido en píxeles.
- *ASCII Offset*: Configura el valor ASCII de la primera ubicación de la RAM de fuentes. Este campo es útil si se conoce que ciertos valores ASCII no son nunca utilizados.

- *Bits per Pixel*: Indica el número de píxeles por cada carácter. Permite una mayor o menor caracterización de los caracteres.
- *Number of Strings*: Indica el número máximo de cadenas que pueden ser almacenadas en RAM. Este valor no puede exceder de 256.
- *Maximum String Length*: Configura la longitud máxima permitida para cada cadena de texto.

3.1.2.9 Video Timing Controller

Los formatos de vídeo emplean señales de sincronismo que permiten describir el flujo de datos de un determinado formato de video. Este bloque se encarga de generar dichas señales de sincronismo a partir del formato de vídeo indicado por el usuario. Este bloque también puede detectar que formato de vídeo está siendo utilizado por un determinado flujo de vídeo.

En el caso de esta aplicación, el bloque será configurado únicamente para la generación de las señales de sincronismo por lo que únicamente necesita ser conectado a la señal de reloj asociada al flujo de vídeo.

La configuración utilizada para este bloque puede observarse en la Figura 3.16:

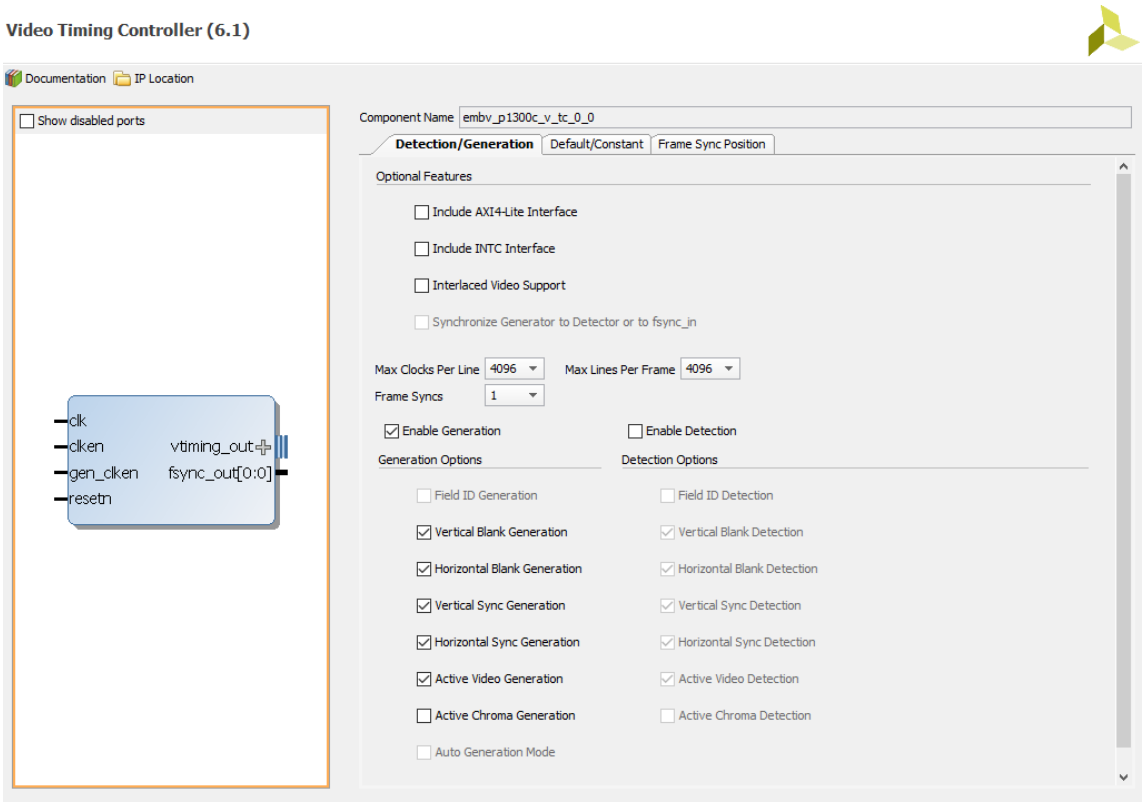


Figura 3.16: Configuración bloque IP Video Timing Controller

En la pestaña *Detection/Generation* se reflejan las principales configuraciones de este bloque.

El apartado *Optional Features* recoge las opciones *Include AXI4-Lite Interface* y *Include INTC Interface* incluidas en muchos de los bloques y ya explicadas anteriormente. En este mismo apartado se incluyen las siguientes opciones:

- *Interlaced Video Support*: Habilita o deshabilita la detección o generación de señales de sincronismo en vídeos con formato interlazado.
- *Synchronize Generator to Detector of to fsync_in*: Cuando está habilitado, el generador se sincroniza automáticamente con el detector o con la señal de entrada *fsync_in*.

El apartado *Max Clocks Per Line* se emplea para indicar el máximo número de píxeles por línea y *Max Lines Per Frame* para indicar el máximo número de líneas por *frame*.

La opción *Frame Syncs* selecciona el número de señales de sincronización independientes generadas a la salida. El valor máximo de este apartado es 16.

La selección de la opción *Enable Generation* habilita la generación de las señales temporales y de sincronización. Esta opción habilita, a su vez, una serie de opciones que permiten seleccionar que señales serán generadas y cuales no:

- *Field ID Generation*
- *Vertical Blank Generation*
- *Horizontal Blank Generation*
- *Vertical Sync Generation*
- *Horizontal Sync Generation*
- *Active Video Generation*
- *Active Chroma Generation*
- *Auto Generation Mode*: Cuando la detección está habilitada, la selección de esta opción permite generar los mismos parámetros de sincronización que los disponibles en la señal de video detectada.

De la misma forma, la selección de *Enable Detection* permite la habilitación de un canal de entrada a partir del cual el bloque podrá detectar los parámetros temporales de dicha señal de video. Esta opción habilita a su vez distintas opciones, similares a las enumeradas anteriormente.

La pestaña *Default/Constant*, mostrada con la configuración utilizada para esta aplicación en la Figura 3.17, permite configurar los aspectos específicos que definen el formato de salida del vídeo y, por tanto, sus parámetros temporales de sincronización.

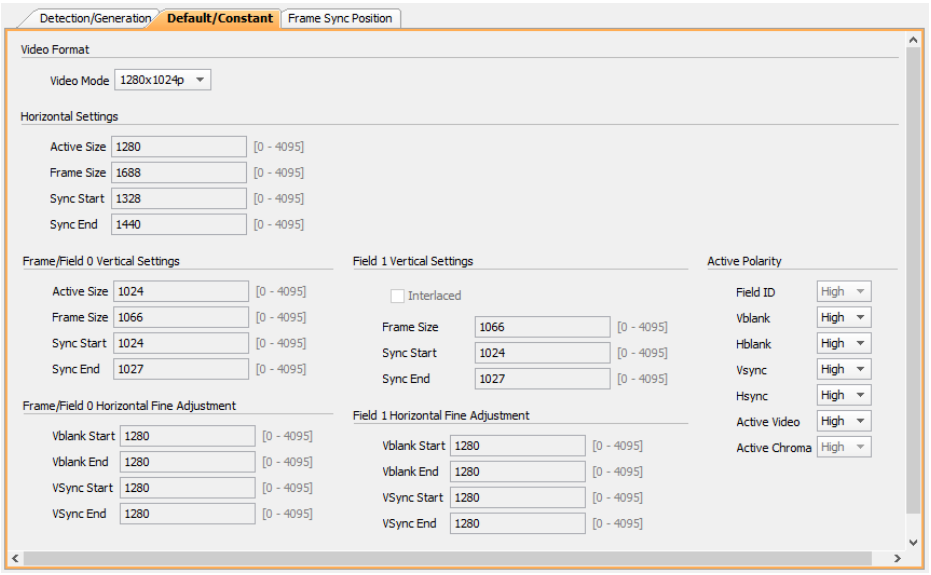


Figura 3.17: Pestaña Default/Constant bloque IP VTC

La configuración *Video Mode* es la opción más importante de esta pestaña ya que permite seleccionar que formato de vídeo se desea emplear. Una vez seleccionado uno de los formatos, el resto de los parámetros de esta pestaña se rellenan automáticamente con el objetivo de cumplir dicho estándar. Si se desea emplear un formato completamente personalizado, se puede seleccionar la opción *Custom* que habilita la edición de todos los apartados de configuración de esta pestaña.

En este caso se ha seleccionado la opción *1280x1024p* que equivale a la resolución *SXGA* proporcionada por el sensor de imagen PYTHON 1300.

3.1.2.10 AXI 4-Stream to Video Out

Este bloque se encarga de combinar el flujo de datos que componen los *frames* capturados por el sensor y las señales de sincronismo generadas por el bloque *Video Timing Controller*. De esta forma, a la salida de este bloque se obtiene una señal de vídeo formada por un conjunto de datos en paralelo y las señales de sincronización.

La configuración de este bloque para la aplicación desarrollada se puede observar en la Figura 3.18.

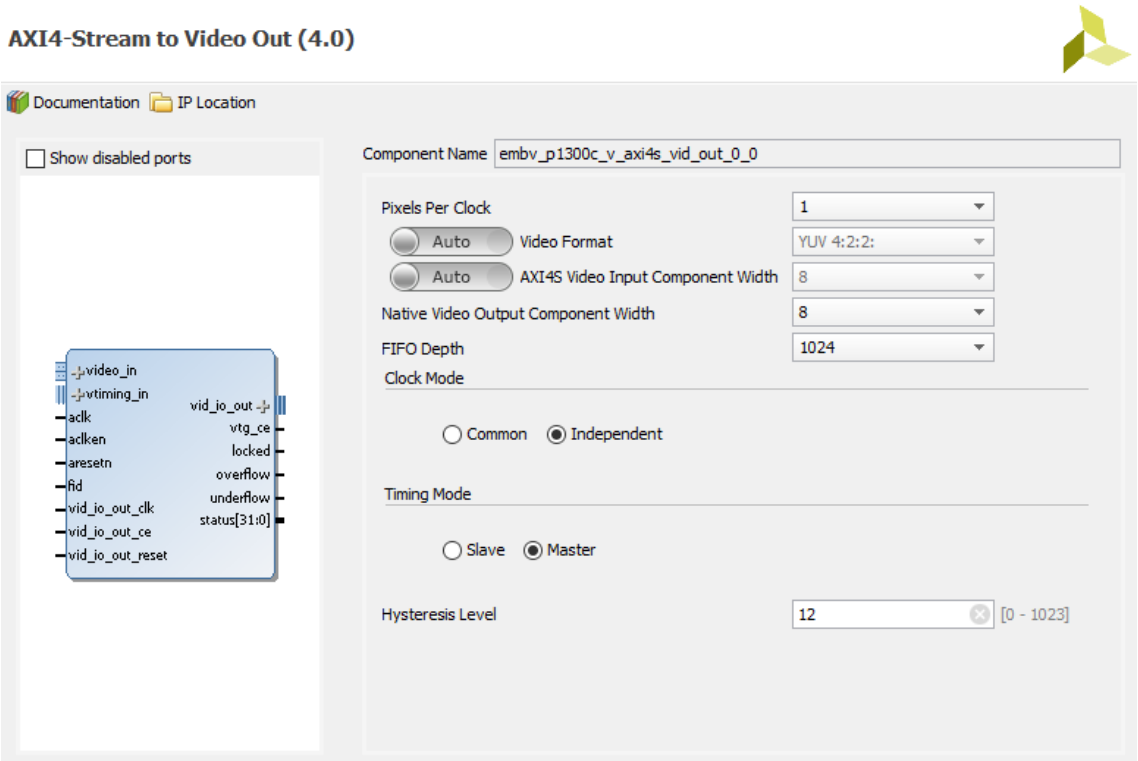


Figura 3.18: Configuración bloque IP AXI4-Stream to Video Out

La opción *Pixels Per Clock* determina el número de píxeles en paralelo que se generan a la salida por cada ciclo de reloj. Los valores permitidos son 1, 2 o 4.

Los parámetros *Video Format* y *AXI4S Video Input Component Width* son detectados automáticamente por la ventana de configuración a partir de las conexiones del bloque e indican que formato de vídeo se tiene a la entrada del bloque y cuantos bits componen cada uno de los canales que conforman el flujo de entrada. De la misma forma, la opción *Native Video Output Component Width* permite indicar el número de bits que compone cada canal a la salida del bloque.

El campo *FIFO Depth* permite seleccionar el tamaño de la memoria FIFO utilizada a la entrada de este bloque. Esta memoria FIFO es necesaria en el caso de que la frecuencia del *stream* de datos de entrada sea menor que la del flujo de salida. En este caso, es necesario almacenar suficientes píxeles por línea de manera que estos puedan ser enviados de forma continua por el flujo de salida.

Sin embargo, en la aplicación desarrollada, la frecuencia del *stream* de datos de entrada (150 MHz) es mayor que la del flujo de salida (108 MHz) por lo que el tamaño de la memoria FIFO es irrelevante en este caso.

El apartado *Clock Mode* permite indicar si se emplea una misma señal de reloj para el flujo de entrada de datos y los datos a la salida de este bloque. En este caso, se ha seleccionado la opción *Independent* ya que la frecuencia a la que llegan los datos de entrada es diferente a la frecuencia de salida deseada.

En *Timing Mode* se permite controlar la relación entre este bloque y el *Video Timing Controller*. En el modo *Slave* el *Video Timing Controller* actúa como esclavo frente a este bloque y es habilitado por este a través de una señal de habilitación. En el modo *Master*, el *Video Timing Controller* es el maestro y el bloque *AXI4-Stream to Video Out* emplea las señales provenientes de este como referencia para la sincronización del vídeo de salida.

La opción *Hysteresis Level* define el número de elementos que deben ser escritos en la memoria FIFO antes de que se comiencen a leer para ser procesados.

3.1.2.11 Avnet HDMI Output (for ADV7511)

Este bloque se encarga de generar un flujo de datos con el formato adecuado integrando las señales de sincronismo. Los datos a la salida de este bloque serán los que reciba el dispositivo ADV7511 encargado de generar las señales diferenciales TMDS que conforman la señal HDMI. Para ello se emplea un puerto externo que, al igual que se ha explicado anteriormente, necesita ser configurado en el archivo *constraints*:

```
set_property IOSTANDARD LVCMOS25 [get_ports {IO_HDMIO_data[0 .. 15]}]
set_property IOSTANDARD LVCMOS33 [get_ports IO_HDMIO_clk]
set_property IOSTANDARD LVCMOS25 [get_ports IO_HDMIO_spdif]
set_property PACKAGE_PIN U14 [get_ports IO_HDMIO_clk]
set_property PACKAGE_PIN L15 [get_ports IO_HDMIO_spdif]
set_property PACKAGE_PIN H15 [get_ports {IO_HDMIO_data[0 .. 15]}]
```

3.1.2.12 ZYNQ7 Processing System

En este bloque se reúne toda la configuración disponible para el dispositivo Zynq. En el apartado *Zynq Block Design*, mostrado en la Figura 3.19, se pueden ver todos los módulos que son configurables en color verde. Seleccionando cada uno de ellos se puede acceder a la configuración de cada uno de los parámetros que los especifican.

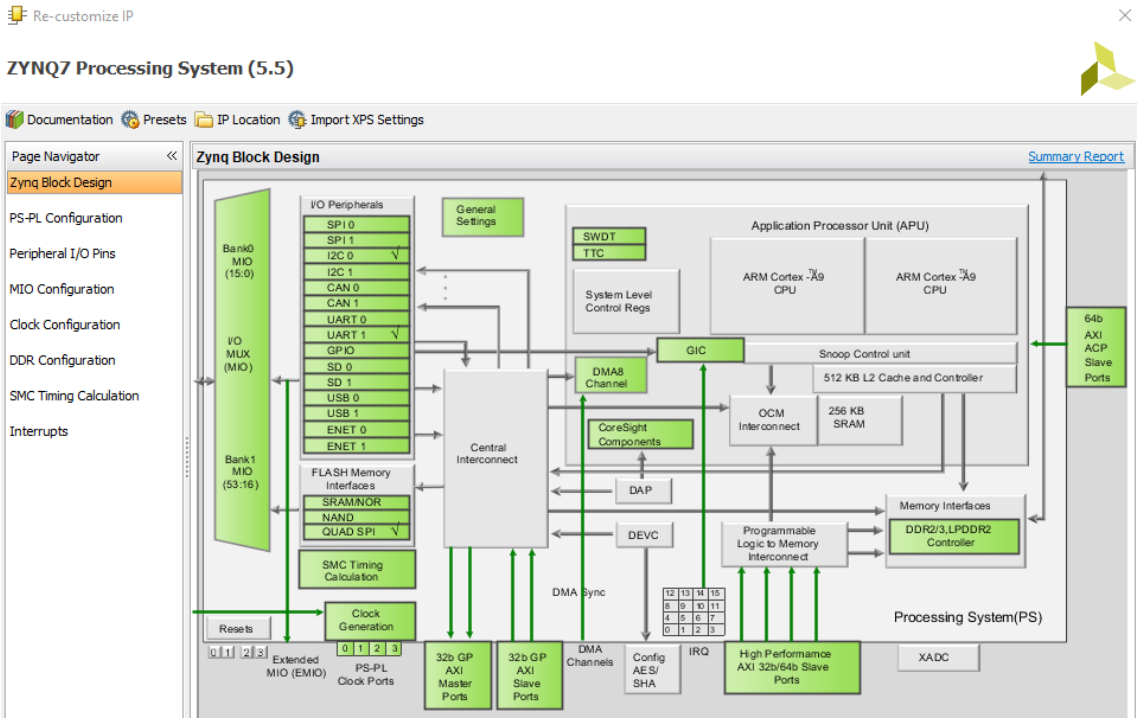


Figura 3.19: Configuración Zynq Block Design

En el apartado *I/O Peripherals* dentro de la pestaña *MIO Configuration* se pueden activar o desactivar los distintos periféricos disponibles en el dispositivo Zynq. Para el caso de esta aplicación, únicamente es necesario activar los periféricos *I2C 0* y *UART 1*, tal y como se puede apreciar en la Figura 3.20.

El periférico *I2C 0* es el encargado de controlar los distintos dispositivos I2C disponibles en la tarjeta *EMBV*. Como se ha explicado anteriormente en esta memoria, todos los dispositivos I2C de esta tarjeta se encuentran direccionados tras un multiplexor I2C por lo que este es el dispositivo conectado directamente a las señales del Zynq.

El periférico *UART 1* es el encargado de controlar el puerto serie conectado al micro-usb de la tarjeta MicroZed. En el apartado *General* de la pestaña *PS-PL Configuration* se puede seleccionar la velocidad de trabajo del puerto serie. En este caso, se va a dejar la velocidad fijada por defecto, 115200 baudios.

El periférico *SD 0* se encuentra conectado al lector de tarjetas micro-SD disponible en la tarjeta MicroZed. Este periférico debe ser habilitado si se desea utilizar este método como modo de programación. A lo largo de este trabajo se va a utilizar un cable JTAG como modo de programación por lo que no se necesita tener activa esta opción.

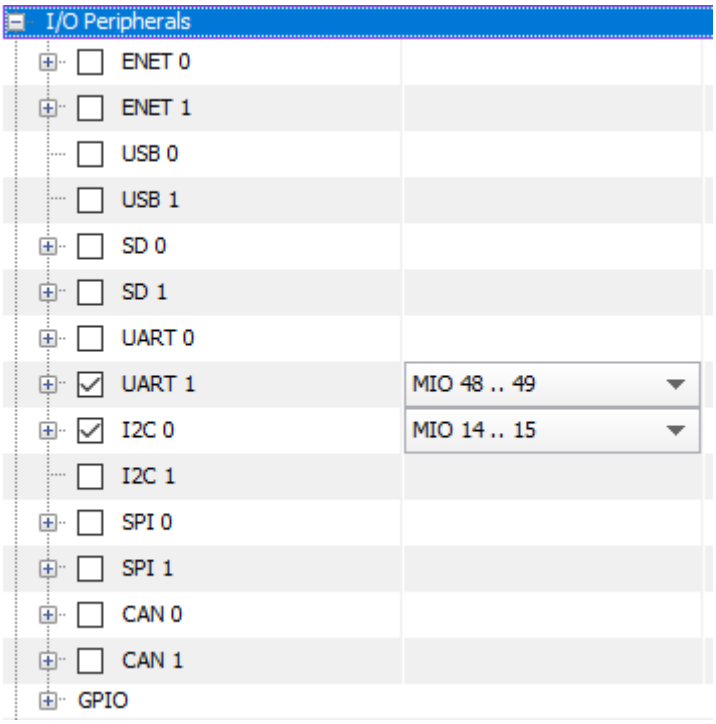


Figura 3.20: Apartado I/O Configuration bloque ZYNQ

Otro de los aspectos importantes en la configuración del dispositivo Zynq es la generación de las distintas señales de reloj que se necesitan en el diseño. Toda la configuración relacionada con las señales de reloj se encuentra en la pestaña *Clock Configuration* y se puede observar en la Figura 3.21.

La frecuencia de la señal de reloj de la CPU se ha dejado por defecto por lo que se ha configurado a la máxima posible, 666 MHz. De la misma forma, la señal de reloj de la memoria DDR también se ha dejado por defecto, configurándose así la máxima admitida de 533 MHz.

Para generar las señales de reloj que se van a utilizar en la parte PL, se deben habilitar estas en el apartado *PL Fabric Clocks*. En esta aplicación se deben generar 3 señales:

- FCLK_CLK0: Configurada a una frecuencia de 75 MHz.
- FCLK_CLK1: Configurada a una frecuencia de 150 MHz.
- FCLK_CLK2: Configurada a una frecuencia de 200 MHz.

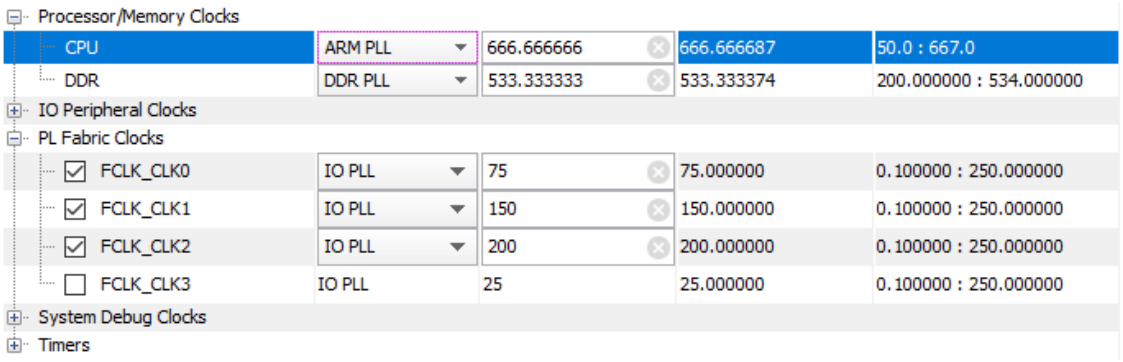


Figura 3.21: Pestaña Clock Configuration bloque ZYNQ

Por último, en la pestaña *PS-PL Configuration* es necesario habilitar, tal y como se puede observar en la Figura 3.22, dos puertos AXI a través de los cuales se realizará la interconexión entre la parte PS y la parte PL. Por un lado, se debe habilitar un puerto maestro de propósito general que se encargará de la gestión de las diferentes interfaces *AXI-Lite* de los bloques IP del diseño. Por otro lado, se habilitará un puerto esclavo de alto rendimiento que se utilizará para el acceso a la memoria DDR a través del protocolo AXI.

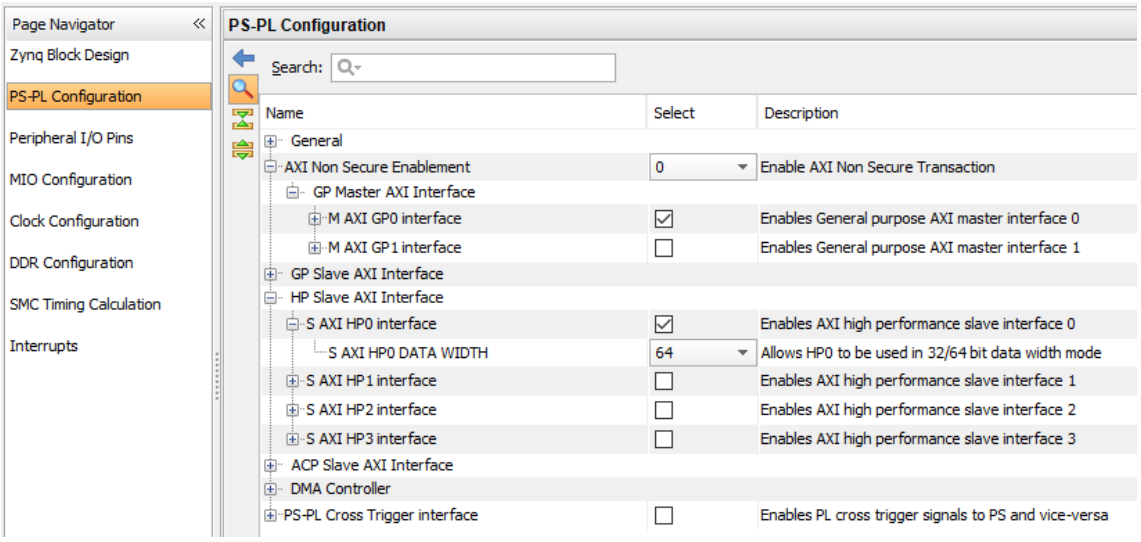


Figura 3.22: Configuración puertos AXI bloque IP ZYNQ

3.1.3 Señales de reloj y conexionado

Las diferentes señales de reloj que se han ido comentando a lo largo de los apartados anteriores se resumen a continuación, indicando para cada una de ellas su identificación, su función y los bloques a los que se asocia:

- FCLK_CLK0: Configurada con la opción *PL Fabric Clocks* a partir del PLL dirigido a los periféricos. Señal a 75 MHz utilizada por el protocolo *AXI-Lite*. Utilizada por todos los bloques IP que emplean este protocolo: *ON Semiconductor PYTHON Camera Receiver*, *AXI VDMA*, *Video On Screen Display* y *ON Semiconductor PYTHON SPI Controller*.
- FCLK_CLK1: Señal de reloj igual a la anterior pero a una frecuencia de 150 MHz. Se emplea como señal de sincronización para el protocolo *AXI4-Stream*. Los bloques que utilizan esta señal son: *Video In to AXI4-Stream*, *Color Filter Array Interpolation*, *RGB to YCrCb Color-Space Converter*, *Chroma Resampler*, *AXI VDMA*, *Video On Screen Display* y *AXI4-Stream to Video Out*.
- FCLK_CLK2: Señal de reloj similar a las anteriores pero configurada a una frecuencia de 200 MHz. Esta frecuencia es requerida por el módulo de la cámara para operar correctamente. Así, los bloques que emplean esta señal de reloj son: *ON Semiconductor PYTHON Camera Receiver* y *Clocking Wizard*.
- clk_out1: Señal de reloj a 108 MHZ obtenida mediante el *IP Core Clocking Wizard* a partir de la señal de reloj FCLK_CLK2. Este bloque es necesario debido a que los PLL usados por el procesador no pueden conseguir con precisión esta frecuencia de reloj. Esta señal de reloj es la asociada al flujo de vídeo con resolución *SXGA* (*Super Extended Graphics Array*) que proporciona el sensor de imagen PYTHON 1300. De la misma forma, esta señal de reloj es utilizada para generar el flujo de vídeo de salida, a través del protocolo HDMI. Así, los bloques que utilizan esta señal de reloj son: *ON*

Semiconductor PYTHON Camera Receiver, *Video In to AXI4-Stream*, *Video Timing Controller*, *AXI4-Stream to Video Out* y *Avnet HDMI Output (for ADV7511)*.

Para conectar las diferentes interfaces esclavas *AXI-Lite* de los bloques *hardware* que componen el diseño con la interfaz maestra AXI de propósito general del dispositivo Zynq es necesario usar un bloque *AXI Interconnect*. Este bloque dispondrá de una interfaz esclava conectada al puerto de propósito general del dispositivo Zynq y 4 puertos maestros, conectados cada uno de ellos a los puertos *AXI-Lite* esclavos de los bloques con dicho puerto activo.

De la misma forma, se va a emplear otro bloque *AXI Interconnect* para conectar los dos puertos AXI maestros del bloque VDMA con el puerto esclavo de alto rendimiento del dispositivo Zynq.

Por último, es necesario añadir 3 bloques *Processor System Reset* que se encargan de producir las señales de *reset* de la parte *hardware* para las señales de reloj *FCLK_CLK0*, *FCLK_CLK1* y *clk_out1*.

3.1.4 Generación archivo bitstream y exportación

Una vez llevado a cabo el diseño de la aplicación, es necesario realizar su síntesis e implementación. Este proceso genera una serie de informes que recogen detalladamente los aspectos relacionados con los tiempos, recursos y consumos eléctricos que utiliza el sistema. El resumen de los recursos utilizados por este primer diseño se muestra en la Figura 3.23. Como se puede observar, estos recursos son relativamente reducidos, lo que permitirá posteriormente implementar y añadir a este diseño un algoritmo que realice un procesamiento de las imágenes capturas por el sensor PYTHON.

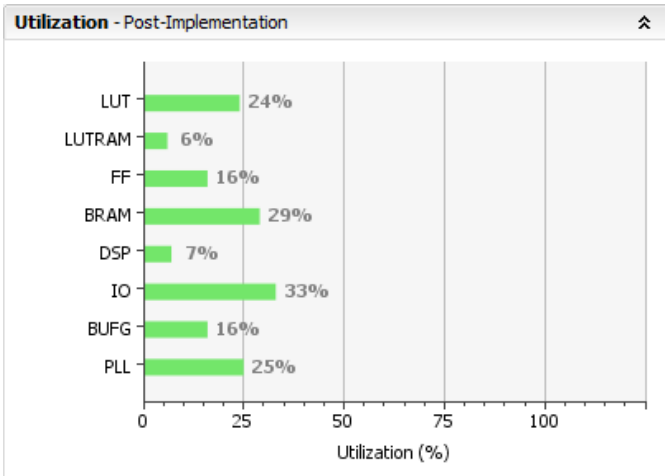


Figura 3.23: Recursos utilizados por la aplicación tras el proceso de implementación

Terminados estos procesos, es posible generar el archivo *bitstream* que recoge toda la información necesaria para la programación de la parte PL del dispositivo Zynq.

Por último, es preciso exportar este archivo de forma que pueda ser accedido desde la herramienta Vivado SDK utilizada para realizar la programación *software* del diseño.

3.2 SDK – Software

Todo el sistema de captura y visualización de imágenes se ha diseñado en *hardware* de forma que el procesador solo se encarga de llevar a cabo la configuración de distintos elementos al inicio de la ejecución, haciendo uso de una aplicación *baremetal*, sin SO.

A partir de funciones que proporcionan los drivers de los distintos bloques HW implementados, se ha realizado la siguiente configuración para la correcta puesta en marcha del ejemplo de captura y visualización de imágenes:

- Inicialización I2C y configuración de la interfaz HDMI
- Configuración cámara
- Configuración *VDMA*
- Configuración *Video On Screen Display*

3.2.1 Bibliotecas utilizadas

Se han empleado una serie de bibliotecas, generadas por los fabricantes, para el control de alguno de los dispositivos que se emplean en este proyecto:

- ***onsemi_python_sw_v3_3***: Biblioteca encargada de facilitar la inicialización y configuración de todos los registros del sensor óptico PYTHON. Dispone de funciones específicas para la escritura y lectura de los registros utilizando el protocolo SPI. Además, proporciona funciones que permiten cambiar parámetros como el tiempo de exposición o la ganancia de forma sencilla.
- ***adv7511***: Se encarga de configurar adecuadamente el dispositivo ADV7511 encargado de la transmisión HDMI.
- ***tca9548***: Gestiona el control del dispositivo Texas Instruments PCA9548A, ubicado en la tarjeta EMBV, encargado de la multiplexación del bus I2C proveniente de la tarjeta MicroZed. Simplifica la selección del canal al que se desea acceder.
- ***pca9534***: Se encarga del control del dispositivo Texas Instruments PCA9534 ubicado en la tarjeta EMBV. La función de este dispositivo es expandir (*I2C I/O Expander*) uno de los canales I2C del dispositivo PCA9548A generando 8 señales I/O (*GPIO*) usadas para el control de diferentes periféricos. Este *driver* permite configurar de forma sencilla la dirección de cada una de las 8 señales y su lectura o escritura.
- ***cat9554***: Inicializa y configura el dispositivo ON Semiconductor PCA9654 disponible en el módulo de la cámara. Este dispositivo es un expansor I2C (*I2C I/O Expander*) que genera 8 señales GPIO, aunque en este caso, únicamente 3 de ellas son utilizadas. Este *driver* simplifica la inicialización del dispositivo y el control de las 3 señales GPIO empleadas para la habilitación de los reguladores necesarios para el correcto funcionamiento del sensor PYTHON.
- ***xaxivdma_ext***: Biblioteca que simplifica la configuración del VDMA para la transmisión de video hacia y desde memoria.
- ***xiicps_ext***: Biblioteca que facilita la escritura y lectura de un determinado registro de un dispositivo I2C.

3.2.2 Inicialización I2C – Configuración HDMI

El chip ADV7511 encargado de la transmisión del flujo de video por HDMI necesita ser inicializado y configurado en tiempo de ejecución. Para ello, emplea el protocolo de comunicación I2C.

De la misma forma, los reguladores del módulo de la cámara también son inicializados a través de este protocolo.

El primer paso para poder utilizar el protocolo I2C es inicializar el *driver*. Para ello, primero se busca su configuración, indicando el dispositivo deseado tal y como figura en el archivo *xparameters.h*, y se inicializa. Una vez inicializado, se realiza un *reset* y se fija la frecuencia de reloj que se desea utilizar en el dispositivo I2C. Esta frecuencia de reloj se indica en hertzios y se ha fijado en 100 KHz.

```
piicps_config = XIicPs_LookupConfig(XPAR_XIICPS_0_DEVICE_ID);
XIicPs_CfgInitialize(pdemo->piicps0, piicps_config, piicps_config-
>BaseAddress);
XIicPs_Reset(pdemo->piicps0);
XIicPs_SetSClk(pdemo->piicps0, 100000);
```

Una vez que se ha inicializado el *driver* I2C, se va a llevar a cabo la configuración del dispositivo expansor pca9534 ubicado en la tarjeta EMBV. Para ello, primero se selecciona en el dispositivo multiplexor el canal al que se desea acceder, *EMBV_IIC_MUX_I2CIO*, que corresponde con el dispositivo expansor. Una vez que se ha seleccionado el dispositivo expansor, se configura si un determinado puerto es de entrada o de salida.

La función *pca9534_set_pins_direction* se encarga de enviar al registro de configuración la información donde se indica si cada uno de los pines es de salida o de entrada. De esta forma, un 1 indica que el pin es de entrada y un 0 que es de salida. En este caso, se configura que únicamente el pin 3 es de entrada, siendo el resto de salida.

Control Signal	Function	Default Value	Direction	Expander Port
HDMI_RST#	HDMI Input Reset	Pulled-High	Output	P0
TP2	Test Point – 2	Open	I/O	P1
HDMI_HPD_CTRL	HDMI Input Hot Plug Detect Control	Pulled-Low	Output	P2
TX_HPD_FPGA	HDMI Output Hot Plug Detect	Pulled-Low	Input	P3
HDMIO_PD	HDMI Output Power Down	Pulled-Low	Output	P4
ETH_RST#	Gigabit Ethernet Reset	Pulled-High	Output	P5
CAM_PWDN_H	Camera Interface Power Down	Pulled-High	Output	P6
TP54	Test Point – 54	Open	I/O	P7

Figura 3.24: Conexiones del dispositivo I2C Expansor [3]

En la Figura 3.24 se puede observar que dicho pin 3 es el encargado de comprobar si existe un dispositivo conectado a la interfaz HDMI de salida de la tarjeta EMBV.

```
tca9548_i2c_mux_select(pdemo->piicps0, EMBV_IIC_MUX_I2CIO);
//Selección del canal expansor
status = 0x08;    //00001000
pca9534_set_pins_direction(pdemo->piicps0, status); //Configuración de
los pines como entrada/salida
```

Para el correcto funcionamiento de la salida HDMI, el primer paso es la deshabilitación de la señal *HDMI Output Power Down* conectada al puerto 4 del expansor I2C. Para ello, se debe seleccionar en el multiplexor el canal asociado al expansor y, a continuación, fijar un valor bajo a dicho pin.

```
tca9548_i2c_mux_select(pdemo->piicps0, EMBV_IIC_MUX_I2CIO);
//Selección del canal expansor
```

```
pca9534_set_pin_value(pdemo->piicps0, 4, 0); //Deshabilitación señal
HDMI Output Power Down
```

El dispositivo ADV7511 encargado del procesado de la señal HDMI de salida se encuentra conectado a uno de los puertos del multiplexor I2C. La configuración de este dispositivo se lleva a cabo escribiendo en ciertos registros determinados valores relacionados con el formato de imagen de entrada y de salida, el color, el tamaño o el sincronismo. Para ello se emplea la función *adv7511_configure* que se encarga de escribir secuencialmente la configuración adecuada en los registros del dispositivo.

```
tca9548_i2c_mux_select(pdemo->piicps0, EMBV_IIC_MUX_HDMIO);
//Selección del dispositivo HDMI de salida en el multiplexor I2C
adv7511_configure(pdemo->piicps0); //Configuración del dispositivo
HDMI de salida ADV7511
```

3.2.3 Configuración del módulo cámara PYTHON

Las siguientes funciones inicializan los bloques *ON SEMICONDUCTOR VITA SPI CONTROLLER* y *ON SEMICONDUCTOR VITA CAMERA RECEIVER* en el diseño *hardware*.

```
onsemi_python_init(pdemo->pPython_receiver, "PYTHON-1300-C",
                   XPAR_ONSEMI_PYTHON_SPI_0_S00_AXI_BASEADDR,
                   XPAR_ONSEMI_PYTHON_CAM_0_S00_AXI_BASEADDR);
onsemi_python_spi_config(pdemo->pPython_receiver, 4);
pdemo->pPython_receiver->uManualTap = 25; // IDELAY setting (0-31)
```

Se realiza un *reset* del dispositivo PYTHON deshabilitando y habilitando el funcionamiento de los reguladores de tensión. Como se ha comentado anteriormente, la señal de habilitación de cada regulador está conectado a un pin de salida del expansor I2C del módulo por lo que únicamente es necesario realizar una escritura I2C para cambiar el nivel de cada señal.

Para ello, el primer paso es seleccionar en el multiplexor I2C de la tarjeta EMBV el canal reservado a la interfaz de la cámara. Una vez seleccionado, se debe inicializar el expansor I2C de la tarjeta PYTHON configurando los puertos donde están conectadas las señales de habilitación (0, 1 y 2) como salida.

A continuación, se aplica el *reset* al sensor PYTHON configurando cada uno de los pines de habilitación a nivel bajo para, finalmente, fijarlos a nivel alto. Cabe destacar que el orden en el que se apagan y se encienden los reguladores es importante ya que se encuentran conectados en cascada, tal y como se puede observar en la Figura 3.25.

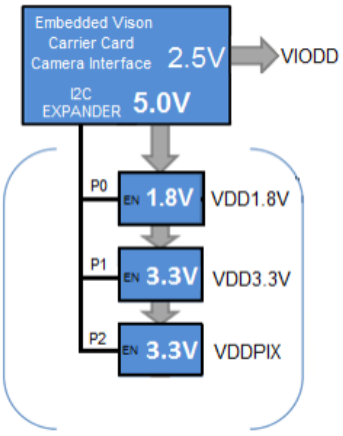


Figura 3.25: Esquema de alimentación del módulo de la cámara [6]

A partir de este momento, los tres reguladores de tensión estarán habilitados generando a su salida los voltajes requeridos para el correcto funcionamiento del sensor de imagen.

```
tca9548_i2c_mux_select(pdemo->piicps0, EMBV_IIC_MUX_CAM); //Selección
del dispositivo I2C disponible en la tarjeta con el sensor PYTHON
cat9554_initialize(pdemo->piicps0); //Inicialización del dispositivo
Expansor I2C disponible en la tarjeta con el sensor PYTHON
usleep(10);
// Se deshabilitan todos los reguladores de tensión que alimentan al
sensor PYTHON
cat9554_vddpix_off(pdemo->piicps0);
usleep(10);
cat9554_vdd33_off(pdemo->piicps0);
usleep(10);
cat9554_vdd18_off(pdemo->piicps0);
usleep(1000);
// Se habilitan todos los reguladores de tensión que alimentan al
sensor PYTHON
cat9554_vdd18_en(pdemo->piicps0);
usleep(10);
cat9554_vdd33_en(pdemo->piicps0);
usleep(10);
cat9554_vddpix_en(pdemo->piicps0);
usleep(10);
```

Por último, el sensor debe ser configurado adecuadamente siguiendo una secuencia de escritura en registros predefinida por el fabricante. Una vez configurado, se habilita el sensor para que realice un envío de las muestras de forma continua. Este sensor dispone de la opción CDS o *Correlated Double Sampling* que permite reducir el ruido de la imagen.

```
onsemi_python_sensor_initialize(pdemo->pPython_receiver,
SENSOR_INIT_ENABLE, 0); //Se realiza la inicialización y configuración
del sensor PYTHON siguiendo una secuencia de escritura en registros
predefinida por el fabricante.
onsemi_python_sensor_initialize(pdemo->pPython_receiver,
SENSOR_INIT_STREAMON, 0); //Se habilita la generación del flujo de
píxeles desde el sensor
onsemi_python_sensor_cds(pdemo->pPython_receiver, 0); //Habilitación
de la técnica de muestreo doble correlacionado (Correlated double
sampling - CDS) que reduce el ruido de la imagen
onsemi_python_cam_reg_write(pdemo->pPython_receiver,
                 ONSEMI_PYTHON_CAM_SYNCGEN_HTIMING1_REG, 0x00300500);
//Configuración del registro que ajusta el sincronismo horizontal
```

3.2.4 Configuración bloque Video-DMA

El bloque VDMA (Video-DMA) se encarga de recibir el flujo de datos de la cámara, almacenarlos en memoria DDR y extraerlos posteriormente, empleando para ello DMA. Tal y como se ha explicado anteriormente, este bloque emplea diferentes *buffers*, 3 en el caso de esta aplicación, que permiten realizan la escritura de uno de ellos mientras otro es leído.

Para la configuración correcta de este bloque es imprescindible conocer una serie de parámetros que permiten indicar cómo deben realizarse las transferencias de datos. En la Figura 3.26 se representa un *buffer* con los parámetros más importantes.

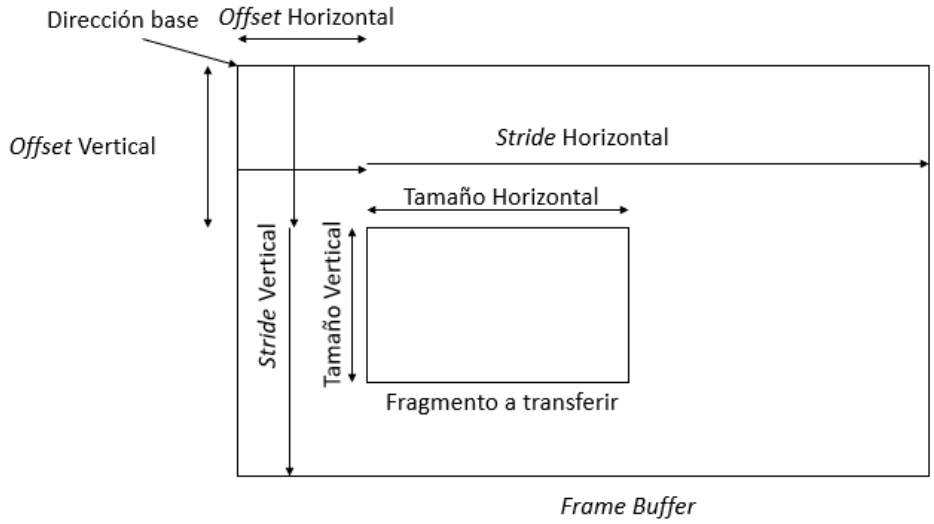


Figura 3.26: Principales parámetros de configuración bloque VDMA

La dirección base representa la dirección de memoria DDR donde comienza un *buffer* cuyo tamaño es fijado por los parámetros *stride* horizontal y vertical. Una vez configurado el tamaño del buffer, es posible configurar una transferencia de datos que suponga una porción de este. Los parámetros *offset* horizontal y vertical permiten fijar cual es el primer píxel que se desea transferir. Por último, los parámetros tamaño horizontal y vertical permiten configurar el número de píxeles a transferir.

Una vez entendidos los principales parámetros de configuración de este bloque, el primer paso necesario para su posterior configuración es la inicialización de sus direcciones y puesta en funcionamiento del bloque con las siguientes instrucciones:

```
paxivdma_config = XAxiVdma_LookupConfig(XPAR_AXIVDMA_1_DEVICE_ID);
XAxiVdma_CfgInitialize(pdemo->paxivdma1, paxivdma_config,
paxivdma_config->BaseAddress);
```

Una vez inicializado, el siguiente paso es la configuración del DMA para el almacenamiento del flujo de datos proveniente del procesado anterior. Para ello, el primer paso es la realización de un *reset* del canal con el objetivo de eliminar otras posibles configuraciones anteriores.

```
XAxiVdma_Reset(pdemo->paxivdma1, XAXIVDMA_WRITE);
```

Para llevar a cabo la configuración del DMA para la escritura de los datos recibidos en DDR se emplea la función *WriteSetup*. Esta función, cuya declaración se muestra a continuación, recibe todos los parámetros explicados anteriormente en este apartado y configura las direcciones base de los 3 *buffers* utilizados en esta aplicación, así como las transferencias que el módulo DMA debe realizar.

```
WriteSetup(XAxiVdma * InstancePtr, u32 BaseAddr, u32 PointNum, u32
EnableCircularBuf, u32 EnableSync, u32 HoriOffset, u32 VertOffset, u32
HoriSize, u32 VertSize, u32 HoriStride, u32 VertStride)
```

Además de los parámetros ya explicados, esta función recoge otros relativos a la sincronización y al modo de funcionamiento del bloque VDMA. Como se explicó en apartados anteriores, este bloque dispone de una serie de señales que permiten sincronizar las lecturas y escrituras de los *buffers*, de modo que estas no se realicen de forma concurrente en un mismo *buffer*. Desde *software* se permite habilitar o deshabilitar estas señales de sincronización mediante el *flag*

EnableSync. Por otra parte, la habilitación del *flag EnableCircularBuf* permite configurar el bloque VDMA para repetir el proceso de forma indefinida. Frente a esta configuración se encuentra el modo *Park* que permite configurar el número de transferencias de datos que deben llevarse a cabo.

En el caso concreto de esta aplicación, se va a configurar un *buffer* del tamaño de la imagen, el modo *buffer* circular y la sincronización, por lo que los parámetros necesarios se muestran a continuación:

```
WriteSetup(pdemo->paxivdma1, 0x18000000, 0, 1, 1, 0, 0, 1280, 1024,
1280, 1024);
```

Cabe destacar que esta función debe tener en cuenta que cada píxel de la imagen está formado por dos bytes por lo que es necesario multiplicar el número de píxeles por dos.

Este bloque VDMA, además de enviar los datos de la imagen por DMA a memoria, también se encarga de acceder a los datos almacenados en memoria y enviarlos a los siguientes bloques que se encargarán de mostrarlos por el puerto HDMI. Para ello, se emplean las instrucciones siguientes:

```
XAxiVdma_Reset(pdemo->paxivdma1, XAXIVDMA_READ);
ReadSetup(pdemo->paxivdma1, 0x18000000, 0, 1, 1, 0, 0, 1280, 1024,
1280, 1024);
```

La función *ReadSetup* es prácticamente idéntica a *WriteSetup* por lo que no es necesario volver a explicar los parámetros que esta emplea. Cabe destacar que la dirección base es la misma utilizada en la configuración de escritura ya que esta es la dirección a partir de la cual se debe realizar la lectura de los datos.

Una vez se ha configurado la transmisión DMA, teniendo instanciado el bloque VDMA con un modo *Dynamic-Master* para el canal de escritura en memoria y un modo *Dynamic-Slave* para el canal de lectura, tal y como se ha explicado anteriormente, ahora únicamente es necesario iniciar la transferencia DMA de forma continua con:

```
StartTransfer(pdemo->paxivdma1);
```

Una vez que se están llevando a cabo las transferencias por DMA, estas seguirán funcionado incluso aunque el procesador no esté ejecutando ninguna instrucción.

3.2.5 Configuración Video On Screen Display

En este caso, el bloque *Video On Screen Display* únicamente se ha configurado con una única capa externa, no estando habilitado el controlador gráfico interno.

Las siguientes funciones llevan a cabo la inicialización de este bloque. La primera de ellas realiza un *reset* de todos los registros y la segunda permite que cualquier cambio efectuado en software se aplique directamente en la configuración de este bloque, sin necesidad de ejecutar una función específica de actualización de registros. Por último, se lleva a cabo la habilitación del bloque.

```
XOSD_Reset(pdemo->posd);
XOSD_RegUpdateEnable(pdemo->posd);
XOSD_Enable(pdemo->posd);
```

Una vez llevada a cabo la inicialización, se configuran las principales características genéricas de la imagen a la salida del bloque. Estas características son independientes de las fijadas para la capa que procesa la imagen proveniente de la cámara. En este caso, se fijan las dimensiones de la imagen a la salida y el color de fondo de las zonas de la imagen donde no hubiera imagen de capa.

```
XOSD_SetScreenSize(pdemo->posd, WIDTH, HEIGHT);//Tamaño 1280x1024 HDMI
XOSD_SetBackgroundColor(pdemo->posd, 0x80, 0x80, 0x80);    //Color de
fondo: gris
```

Por último, se configuran las características de la capa que recibe la imagen del sensor óptico.

```
XOSD_SetLayerPriority(pdemo->posd, 0, XOSD_LAYER_PRIORITY_0);
      //Prioridad de capa: 0
XOSD_SetLayerAlpha(pdemo->posd, 0, 1, 0xFF);    //Parámetro Alpha: 255
XOSD_SetLayerDimension(pdemo->posd, 0, 0, 0, WIDTH, HEIGHT);
//Dimensiones imagen: 1280x1024
XOSD_EnableLayer(pdemo->posd, 0);   //Habilitación de la capa
```

Cabe destacar que la capa utilizada tiene la misma dimensión que la imagen de salida del bloque lo que supone que, en ningún momento se observarán los píxeles de fondo.

Este bloque no sería necesario en este primer diseño puesto que, según como se ha configurado, no añade nada a la imagen que recibe a la entrada. Sin embargo, en diseños posteriores se hará uso del controlador gráfico interno para incorporar elementos a la imagen.

3.3 Ejecución de la aplicación

En este apartado se detalla el conexionado necesario para la ejecución de la aplicación desarrollada, el proceso de programación y los resultados obtenidos.

3.3.1 Conexionado

Los dispositivos y conexiones empleados para la ejecución de esta aplicación se muestran de forma gráfica en la Figura 3.27.

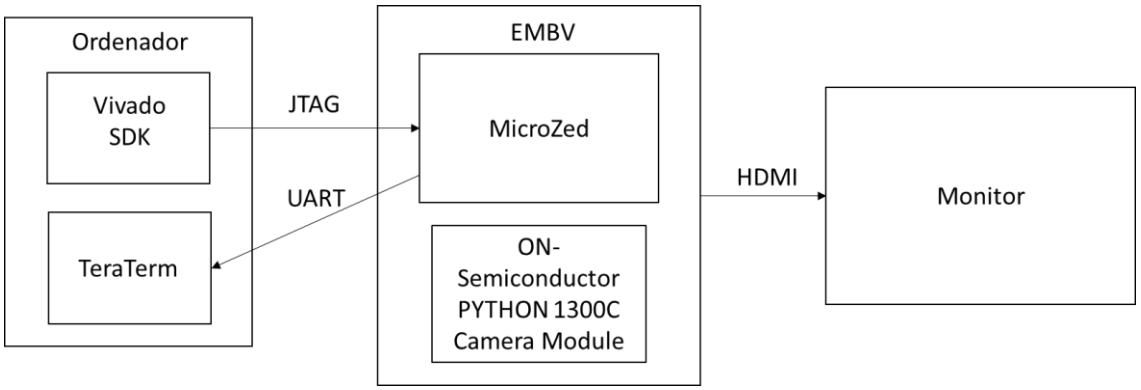


Figura 3.27: Conexionado comprobación de funcionamiento de la aplicación de captura y visualización de imágenes

Los primeros elementos a conectar son la tarjeta de desarrollo MicroZed y la tarjeta de expansión *EMBV*. Estas tarjetas son conectadas entre sí empleando los conectores

MicroHeaders disponibles en ambas. A continuación, el módulo con el sensor PYTHON es conectado a la tarjeta de expansión *EMBV* a través de la interfaz PCIe.

Una vez conectadas todas las tarjetas, es necesario conectar el conjunto a un monitor por el cual se mostrará la señal de video capturada por el sensor PYTHON. Esta conexión se realiza a treves del puerto HDMI de salida de la tarjeta *EMBV* mediante un cable micro-HDMI.

Para la comunicación del puerto serie, se conecta un cable micro-USB al puerto *UART* disponible en la tarjeta MicroZed. Por último, a esta tarjeta se conectará el dispositivo JTAG o la tarjeta microSD, dependiendo del modo de programación del dispositivo ZYNQ elegido.

3.3.2 Programación y ejecución

Una vez que el código se ha compilado sin errores, la herramienta Vivado SDK genera un archivo ejecutable con extensión *elf*. Este archivo será el que se ejecutará en el procesador ARM que incluye el dispositivo ZYNQ. Sin embargo, antes de descargar y ejecutar dicho archivo es necesario realizar la programación de la FPGA con el archivo *bitstream* generado anteriormente por Vivado.

La tarjeta MicroZed puede ser programada usando diferentes métodos como, por ejemplo, una tarjeta microSD o un dispositivo JTAG. Por ello, es necesario configurar una serie de *jumpers* disponibles en la tarjeta MicroZed para indicar la forma de programación. En todas las aplicaciones desarrolladas en este trabajo la programación de la tarjeta MicroZed se va a realizar usando un dispositivo JTAG para lo cual los *jumpers* deben estar colocados tal y como se muestra en la Figura 3.28.

Figura 3.28: Configuración jumpers para el uso de un dispositivo JTAG

Una vez programada la lógica *hardware* y cargado el archivo ejecutable, el programa realiza todas las configuraciones explicadas anteriormente. Cuando estas se han llevado a cabo, en el monitor se puede apreciar en tiempo real las imágenes que están siendo capturadas por la cámara. En la Figura 3.29 se pueden observar los distintos elementos utilizados durante la ejecución de la aplicación.

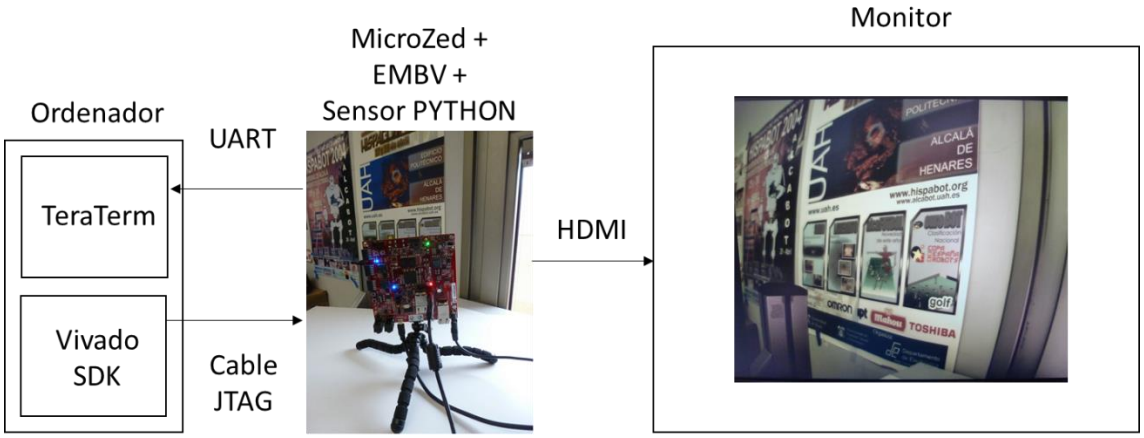


Figura 3.29: Ejecución de la aplicación desarrollada

Para conocer la frecuencia de generación y emisión de las imágenes observadas, se ha diseñado un pequeño bloque en VHDL, mostrado en la Figura 3.30, que cuenta el número de ciclos de reloj que hay entre dos flancos de subida de la señal *user* del bus *AXI-Stream*. Esta señal, como se ha comentado anteriormente, se pone a nivel alto durante el tiempo que el flujo de vídeo transmite el primer píxel de una imagen, señalando así el comienzo de un *frame*. A partir del número de ciclos contados y la frecuencia del bus, se puede calcular el número de fotogramas por segundo.

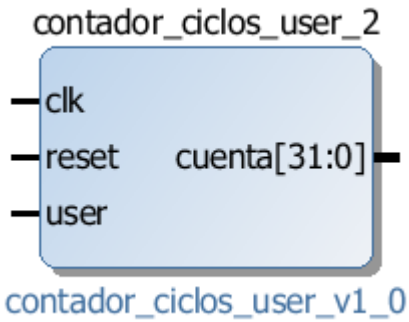


Figura 3.30: Bloque VHDL contador de ciclos de reloj entre dos flancos de la señal user

Así, en relación a las características del vídeo de salida, este se muestra a 60 *frames* por segundo. Sin embargo, el proceso de generación de imágenes es más lento, consiguiendo una tasa de refresco de, aproximadamente, 37 fotogramas por segundo. Esta diferencia de tasas es controlada por el bloque VDMA, de forma que, como la escritura es más lenta, los *buffers* que no están siendo escritos se repiten, leyéndose en más de una ocasión.

Capítulo 4: Generación y visualización de un histograma en tiempo real

En este capítulo se pretende calcular y visualizar, superponiéndolo en la imagen y en tiempo real, el histograma asociado a las imágenes capturadas por el sensor PYTHON. Para ello, se realizará un diseño *hardware* que posteriormente será añadido al diseño en Vivado descrito en el capítulo anterior.

4.1 Histograma de una imagen

Un histograma es una representación gráfica que muestra la distribución de los distintos valores de píxel de una imagen. Así, en el caso de una imagen en escala de grises, un histograma recoge para cada valor posible de luminancia, cuantos píxeles tienen un mismo valor.

Un histograma es representado mediante el empleo de múltiples barras verticales, una para cada posible valor de luminancia. Estas barras tienen una anchura fija y su altura depende del número de coincidencias del valor de píxel que representan. En la parte de la izquierda se muestran los valores asociados a los niveles más oscuros, cuyo valor asociado de luminancia es bajo y, progresivamente a la derecha, se muestran los niveles más claros, con niveles de luminancia más elevados.

Además, con el objetivo de poder apreciar correctamente un histograma, este suele ser normalizado, haciendo que el valor de píxel más frecuente ocupe todo el eje vertical.

4.2 Implementación en Vivado HLS

Para llevar a cabo el diseño *hardware* se ha empleado la herramienta HLS incluida dentro del conjunto del entorno de desarrollo Vivado.

4.2.1 Vivado HLS

La herramienta Vivado HLS o *Vivado High Level Synthesis* es un *software* diseñado por *Xilinx* que permite describir en lenguaje de alto nivel como C o C++ una determinada función y posteriormente sintetizarla, permitiendo implementarla posteriormente en un dispositivo de HW programable. Así, esta herramienta libera al programador de realizar la programación empleando lenguajes de bajo nivel como VHDL o Verilog. Esto permite una mayor velocidad de diseño, tests y una gran flexibilidad en el desarrollo de diseños *hardware*. Por el contrario, la solución obtenida puede ser menos eficiente que si se desarrollara en un lenguaje de descripción de HW directamente. El avance de los algoritmos de inferencia HW supone que esta posible desventaja sea poco importante en la mayoría de ocasiones.

La programación del algoritmo se lleva a cabo mediante la creación de una o varias funciones que ejecutan la funcionalidad deseada. La función con mayor jerarquía de todas ellas (denominada *top*) tiene como argumentos de entrada las señales y buses que comunicarán la aplicación *hardware* con otros elementos del diseño completo, ya sea a través de una interfaz AXI o con señales HW *ad-hoc*.

Con el objetivo de permitir al programador un mayor control de cómo el diseño es implementado en *hardware*, HLS permite introducir directivas o *pragmas*. Estas directivas optimizan el diseño permitiendo, por ejemplo, indicar qué tipo de recurso *hardware* debe

implementar una determinada operación o con qué nivel de paralelización y, por tanto, con qué cantidad de recursos, debe ejecutarse un determinado bucle.

Para poder comprobar el correcto funcionamiento de la aplicación desarrollada, se debe crear un banco de pruebas o *testbench*. Este *testbench* es simula el resto del sistema a través de una o varias funciones programadas en C/C++ en el cual se integrará el diseño *hardware*. De esta forma, el *testbench* proporciona todas las señales de entrada que la función *top* espera recibir y recoge los datos de salida que este proporcione. Estos datos podrán ser analizados posteriormente por el programador para comprobar si el funcionamiento de la aplicación es el esperado.

Para llevar a cabo la depuración del diseño, este programa provee diferentes herramientas que permiten comprobar su correcto funcionamiento en diferentes niveles:

- Simulación C: Se emplea para comprobar el correcto funcionamiento de la aplicación a nivel C. Esta simulación ejecuta la función de *testbench* que, a su vez, llama a la función *top* que se desea implementar en *hardware*. Con el objetivo de encontrar los errores que se puedan producir, esta simulación puede ser ejecutada paso a paso empleando un entorno de depuración o *debugger*.
- Síntesis C: Esta opción realiza el sintetizado de la función *top* y genera un resumen con los detalles temporales y de uso de recursos necesarios para su implementación. Estos datos permiten conocer, para cada operación llevada a cabo, una estimación de cuáles son los recursos y la latencia que introduce a la aplicación. De esta forma, el programador puede conocer cuáles son las partes del algoritmo que más afectan al rendimiento y área utilizada en su posterior implementación. Cabe destacar que el proceso de sintetizado tiene en cuenta las directivas o *pragmas* introducidos por el usuario ya que estas son las utilizadas por este para indicar como la aplicación debe ser sintetizada.
- Cosimulación C/RTL: Esta herramienta lleva a cabo una simulación de la aplicación utilizando el *testbench* y la implementación RTL. Esta simulación, mucho más lenta que la simulación C, devolverá los resultados esperables una vez la aplicación sea implementada en una FPGA.

Una vez realizados todos los pasos indicados, Vivado HLS se encarga de realizar la exportación del bloque HW para que, posteriormente, pueda ser emplazada como un bloque IP más dentro del repositorio de Vivado.

4.2.2 Codificación en C++

En los siguientes apartados se explica detalladamente los aspectos más importantes del algoritmo implementado en HLS.

4.2.2.1 Función *top*

El bloque HLS va a recibir las imágenes a procesar como un flujo de píxeles codificados en formato YUV 4:2:2. Con esta codificación, cada píxel va a tener dos componentes, luminancia y crominancia, de 1 byte cada una de ellas. La salida de este bloque será igualmente un flujo de píxeles con el mismo formato.

La clase *hls::stream* es el objeto utilizado en HLS para modelar un flujo de datos. Esta clase de datos es implementada como una memoria FIFO de un único valor. El tipo de datos del flujo *AXI4-Stream* es indicado mediante la estructura *ap_axi(u)* que reúne las distintas señales que lo componen. Esta estructura puede ser configurada según las necesidades de la aplicación.

En este caso, como los píxeles siempre toman valores positivos, se emplea el tipo *ap_axiu* con un tamaño de datos de 16 bits (1 byte para cada canal) y un *bit* para el canal *user*.

```
void procesado_imagen(hls::stream<ap_axiu<16,1,1,1> >&video_in,
hls::stream<ap_axiu<16,1,1,1> >&video_out)
```

Para que ambos punteros sean implementados posteriormente como buses *AXI*, es necesario indicárselo a la herramienta HLS mediante el *pragma HLS INTERFACE* tal y como se muestra a continuación.

```
#pragma HLS INTERFACE axis port=video_in
#pragma HLS INTERFACE axis port=video_out
```

El primer elemento indica el tipo de interfaz *AXI* a implementar. La palabra *axis* indica a HLS que la variable indicada por *port* debe ser implementada como un bus *AXI4-Stream*.

El cálculo de un histograma requiere que una imagen sea procesada completamente antes de poder visualizarlo. Sin embargo, debido al procesamiento en forma de flujo, es necesario generar un píxel de salida para cada píxel que se ha leído. Por ello, en esta aplicación se va a calcular el histograma de la imagen que se está recibiendo y se va a superponer sobre esta el histograma del *frame* anterior.

Con el objetivo de facilitar la comprensión del código desarrollado, a continuación se muestra un *pseudocódigo* con los principales elementos que lo conforman:

```
void procesado_imagen(hls::stream<ap_axiu<16,1,1,1> >& video_in,
hls::stream<ap_axiu<16,1,1,1> >& video_out)
{
      #pragma HLS dataflow
      //Declaración de variables
      static ap_uint<21> hist[256]; //Vector donde se almacena el
histograma que se está calculando
      static ap_uint<10> hist_ant[256]; //Vector donde se almacena el
histograma del frame anterior
      hls::AXIvideo2Mat(video_in, imagein);     //Conversión de AXI-
Stream a tipo Mat
      for(int i = 0; i < 1024; i++)
      {
            for(int j = 0; j < 1280; j++)
            {
                  #pragma HLS PIPELINE II = 1
                  //Cálculo del histograma actual
                  //Visualización del histograma anterior
            }
      }
      for(int i=0; i< 256; i++)
      {
            //Búsqueda del valor máximo del histograma
      }
      //Normalización del histograma a partir del valor máximo y el
tamaño máximo (1024 píxeles)
      for(int i=0; i< 256; i++)
      {
            //Cálculo del histograma normalizado
            //Inicialización del histograma para la siguiente ejecución
      }
```

```
      hls::Mat2AXIvideo(imageout, video_out);   //Conversión del tipo
Mat a AXI-Stream
}
```

La función *hls::AXIVideo2Mat* es una función desarrollada por *Xilinx* que convierte un flujo de píxeles recibido por un bus *AXI4-Stream* en un formato *hls::Mat* que representa una imagen en la biblioteca *HLS Video*. Así, esta función se encarga de sincronizar el flujo de datos, leyendo las señales *user* y *last* que indican el principio de una imagen y el final de cada una de sus líneas respectivamente.

El tipo *hls::Mat* no debe confundirse con el tipo *Mat* de OpenCV. El tipo *Mat* de HLS no almacena una imagen en forma de matriz, sino que es implementado como un flujo de píxeles. Por lo tanto, no se pueden realizar accesos a elementos aleatorios, sino únicamente lecturas y escrituras secuenciales.

De forma complementaria, la función *Mat2AXIVideo* realiza el procedimiento contrario, convirtiendo en formato *AXI4-Stream* la imagen generada con el histograma superpuesto. Así, esta función se encarga de generar un pulso en la señal *last* del bus *AXI4-Stream* cada vez que se envía por el bus de datos un píxel de final de línea y un pulso en la señal *user* cuando se envía el primer píxel de una imagen.

En relación a los *pragmas* utilizados en este código, se destacan 2:

- *DATAFLOW*: Esta directiva habilita una segmentación a nivel de función. Esto permite que las diferentes funciones que conforman un diseño no tengan que esperar a que la función anterior termine completamente su ejecución, sino que puede comenzar tan pronto como disponga de los datos de los que dependa. De esta forma se logra que las funciones sean ejecutadas con una cadencia lo más pequeña posible. La aplicación de este *pragma* permite aumentar el rendimiento de la aplicación desarrollada, pero implica una serie de limitaciones como, por ejemplo, la imposición de un modo de diseño donde se tenga un único productor y un único consumidor.
- *PIPELINE II=1*: Este *pragma* permite indicar al sintetizador que se desea que esta parte del código sea segmentada de forma que cada iteración del bucle pueda comenzar un ciclo después de la ejecución del ciclo anterior. La segmentación de bucles permite que diferentes operaciones sean realizadas concurrentemente, logrando que el *hardware* utilizado sea utilizado en todo momento, no teniendo que esperar a que termine la iteración actual para comenzar la siguiente.

El resto del código que compone esta función puede ser dividido en dos partes claramente diferenciadas: la generación del histograma y la visualización de este sobre la imagen capturada.

4.2.2.2 Generación del histograma

Para generar el histograma, el primer paso es acceder al valor de luminancia de cada uno de los píxeles. Para ello, cada píxel recibido es leído de la clase *hls::Mat* con el operador *>>* y almacenado en una variable de tipo *hls::Scalar*. Una variable escalar es utilizada en HLS para representar un único píxel. La declaración y la lectura de un píxel se realiza tal y como se muestra a continuación:

```
hls::Scalar <2,unsigned char> pixel;      //Declaración de variable de
tipo escalar
```

```
imagein >> pixel; //Lectura de un píxel de una variable de tipo
hls::Mat
```

La declaración de la variable *pixel* se define de tipo *scalar* con 2 canales donde cada uno de ellos ocupa un *byte* de datos. Por último, el acceso a cada canal de datos se realiza con la clase *val[num_canal]* tal y como se muestra a continuación:

```
y=pixel.val[0]; //Acceso canal luminancia
c=pixel.val[1];//Acceso canal crominancia
```

Una vez se conoce el valor de luminancia, se procede al cálculo del histograma. El histograma será almacenado en un vector de 256 valores, donde cada uno de ellos recogerá el número de píxeles en una imagen para un determinado valor de luminancia.

El mayor valor que puede tomar un elemento del histograma se dará cuando todos los píxeles que conforman una imagen tengan el mismo valor de luminancia. En ese caso, el valor será el número de píxeles de la imagen que, en este caso, es 1310720.

Para aprovechar los recursos disponibles, se va a utilizar la biblioteca *ap_int* que permite definir el número de *bits* utilizados para almacenar una determinada variable. Si, como en este caso, solo se trabaja con números positivos, se puede utilizar *ap_uint* que permite evitar tener que almacenar el *bit* de signo. En el caso del histograma, el máximo valor que puede llegar a ocupar un dato son 21 *bits*.

La variable además ha sido declarada de tipo estática utilizando la etiqueta *static*. Una variable estática es inicializada en el momento de implementación del bloque y, además, mantiene su valor entre las diferentes ejecuciones de la función. En este caso, los elementos del histograma deben ser inicializados a 0 antes de realizar la acumulación por lo que en este caso se ahorran algunos ciclos de reloj en la primera ejecución.

```
static ap_uint<21> hist[256]; //Declaración del vector donde se
almacenará el histograma
#pragma HLS array_partition variable=hist complete
```

El *pragma array_partition* permite indicar a la herramienta HLS como debe implementar los elementos de un vector en memoria. Aplicar la etiqueta *complete* a un vector permite descomponer el *array* en registros individuales. Esto permite evitar los cuellos de botella que se producen al realizar múltiples lecturas y escrituras en un bloque RAM.

4.2.2.2.1 Cálculo del histograma actual

Un histograma se puede calcular fácilmente leyendo el valor del histograma asociado al píxel leído e incrementando en una unidad su valor.

```
hist[val]=hist[val]+1;
```

Sin embargo, esta operación no puede ser llevada a cabo en un ciclo de reloj, ya que es necesario primero leer el valor actual en el histograma, a continuación incrementar ese valor en una unidad y, finalmente, almacenar el valor resultante en el mismo elemento del intervalo. Esto supone que, al segmentar, se necesiten 2 ciclos de reloj antes de poder comenzar la siguiente iteración ya que, si un píxel es igual al anterior, el valor leído debe ser actualizado antes de que la siguiente iteración pueda leerlo e incrementarlo de nuevo.

Para evitar lo anterior, se ha utilizado un algoritmo que comprueba si el valor de luminancia leído es igual al del píxel anterior. Si es igual, se incrementa en una unidad un contador cuyo valor actual es el valor del histograma para ese elemento, no realizando ninguna lectura ni

escritura en el vector histograma. Si, por el contrario, el píxel leído tiene un valor distinto al anterior, se sobrescribe el valor acumulado en el elemento asociado al valor anterior y se lee el valor del histograma para el elemento asociado al elemento actual, incrementándolo en una unidad y almacenándolo en el contador.

En la Figura 4.1 se muestra un esquema donde se puede observar gráficamente el funcionamiento de este algoritmo.

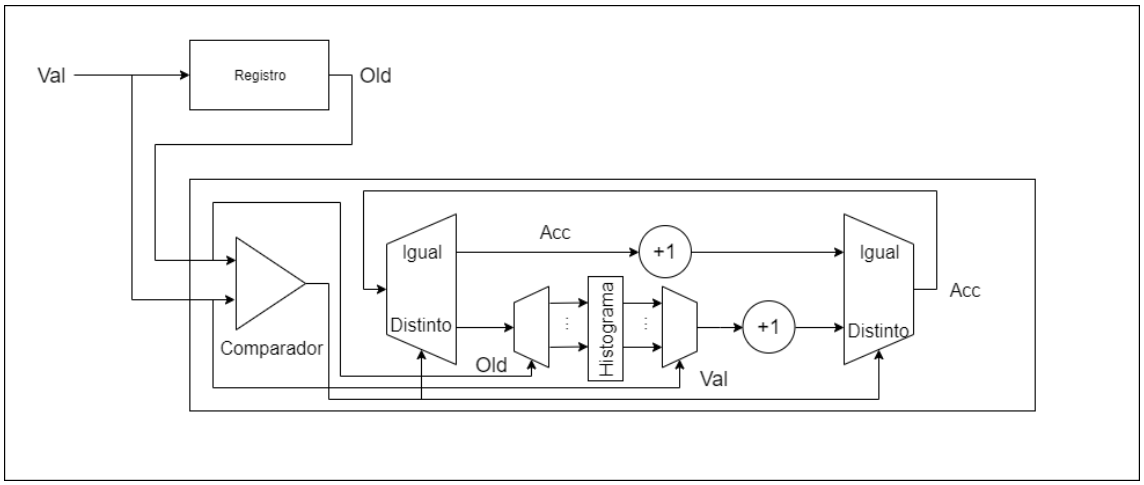


Figura 4.1: Esquema algoritmo cálculo del histograma

Utilizando el método anterior, se evita que dos iteraciones consecutivas tengan dependencia entre sí, ya que nunca accederán al mismo elemento del vector histograma.

A continuación, se muestra el código utilizado para el cálculo del histograma del *frame* actual.

```
val = y;    //Se lee el valor de luminancia
if (old==val) //Se comprueba si el valor leído es igual al anterior
{
      acc++; //Si es igual se incrementa en una unidad
}
else //Si el valor leído es distinto al anterior
{
      hist[old]=acc; //Si no es igual, se sobrescribe el valor
almacenado en el histograma para el valor anterior
      acc=hist[val]+1; //Se lee el valor actual del histograma para el
valor leído, se incrementa en una unidad y se almacena en la variable
temporal acc.
}
old = val; //Se actualiza el valor anterior
```

4.2.2.2.2 Búsqueda del valor máximo y normalización del histograma

Una vez se han recibido todos los píxeles de una imagen, se debe normalizar el histograma para que posteriormente pueda ser representado superpuesto a la imagen siguiente. De esta forma, el mayor valor del histograma se representará con una barra de altura igual al número de filas de la imagen, 1024. El resto de valores tendrán una altura proporcional truncada a un valor entero de píxeles.

El histograma normalizado al número de filas de la imagen se almacena en un *array* estático de 10 *bits* por elemento (valores de 0 a 1023). El vector debe ser declarado estático ya que

debe mantener sus valores entre dos ejecuciones consecutivas de la función *top* de forma que la siguiente ejecución pueda leerlos y visualizarlos sobre la imagen.

```
static ap_uint<10> hist_ant[256];
```

Para llevar a cabo la normalización, el primer paso es obtener el mayor valor del histograma. Para ello, se recorre el *array* elemento a elemento y se comprueba si el dato leído es mayor que el máximo encontrado.

Una vez obtenido el máximo, se calcula el factor de normalización que se obtiene a partir del número de filas de la imagen, 1024, dividido entre el valor máximo del histograma. A continuación, se multiplica el factor calculado por cada elemento del histograma y se almacena el resultado en el vector *hist_ant* que será leído y visualizado en el siguiente *frame*. Por último, se inicializa a 0 el vector *hist* para prepararlo para la siguiente ejecución de la función. La inicialización de la variable en este punto, en vez de en la parte inicial del algoritmo, permite reducir el número de ciclos necesarios en la ejecución de este, ya que en este punto se pueden aprovechar los ciclos ya utilizados para el cálculo de la normalización.

```
hist[old]=acc; //Actualización del histograma con el último valor de
píxel
for(int i=0; i< 256; i++) //Se recorre el histograma completo
{
      if (hist[i]>max) //Se comprueba si el elemento leído es mayor
que el máximo encontrado
      {
            max=hist[i]; //Si es mayor, se almacena
      }
}
temp=constante/max; //Cálculo del factor de normalización: 1024/valor
máximo en histograma
for(int i=0; i< 256; i++) //Se recorre el histograma completo
{
      hist_ant[i]=hist[i]*temp; //Se aplica el factor de normalización
y se almacena en hist_ant
      hist[i]=0; //Se inicializa a 0 cada elemento del histograma para
la siguiente ejecución de la función
}
```

Cabe destacar que las operaciones de división y multiplicación se han realizado utilizando números en coma fija, empleando la biblioteca *ap_ufixed*. Esta notación permite elegir el tamaño de palabra de una determinada variable y, posteriormente, destinar un número de *bits* para la parte entera y otro para la parte decimal. El empleo de este tipo de notación para el cálculo de operaciones matemáticas supone una menor precisión en los resultados obtenidos pero reduce el número de recursos y aumenta el rendimiento de la aplicación.

4.2.2.3 Visualización del histograma

Para realizar la visualización en tiempo real, a la vez que se está recibiendo la imagen y se está calculando su histograma, se va a generar una imagen de salida donde se muestre la imagen actual y, sobre esta, el histograma del *frame* anterior. El histograma representado se mostrará por medio de barras blancas de 5 píxeles por elemento, aprovechando así todo el ancho de la imagen (5*256 = 1280) y cuya altura vendrá dada por la normalización explicada en el apartado anterior.

Los píxeles de la imagen son recibidos por filas desde la superior a la inferior. Por ello, es necesario recorrer los valores del histograma y comprobar para cada píxel si se debe

mantener el valor leído o si se debe sustituir por un píxel de color blanco que conforme parte de la representación del histograma.

El código utilizado para implementar la visualización del histograma se muestra en el siguiente fragmento de código.

```
if (hist_ant[ind] < (1023-i)) //Se lee el elemento del histograma
calculado en el frame anterior y se comprueba si es menor que el
número de fila asociado al píxel que debe generarse
{ //Si es menor, se configura el píxel con los valores del píxel leído
      pixel2.val[0]=y; //Canal de luminancia igual al valor de
luminancia del píxel leído
      pixel2.val[1]=c; //Canal de crominancia igual al valor de
crominancia del píxel leído
}
else //Si es mayor o igual, se configura el valor del píxel de color
blanco
{
      pixel2.val[0]=255;
      pixel2.val[1]=128;
}
a++; // siguiente columna      1280 / 256 => 5 pixeles por valor de
histograma
if(a==5) //Se comprueba si se debe acceder al siguiente elemento del
histograma
{
      a=0;
      ind=ind+1; //Se incrementa el índice que controla el elemento a
leer del histograma
      if(ind==256) //Se comprueba cuando se llega al último píxel de
cada línea
      {
            ind=0; //Se inicializa a 0 el índice que se emplea para
recorrer el histograma
      }
}
```

Una vez que el valor de los dos canales que conforman un píxel ha sido configurado, se escribe este en el tipo *hls::Mat* para su posterior conversión a flujo de datos de tipo *AXI4-Stream*.

```
imageout << s2;
```

4.2.3 Comprobación de funcionamiento: testbench

Una vez se ha llevado a cabo el diseño del sistema, es necesario comprobar su correcto funcionamiento antes de proceder a su síntesis y exportación para su posterior incorporación al sistema completo. Para ello, se ha generado un *testbench* que simula el funcionamiento del resto del sistema al que se acoplará este bloque *hardware* diseñado en HLS. De esta forma, el banco de pruebas se encarga de proveer de datos a la entrada del diseño *hardware*, llamar a la función *top* que implementa el diseño y leer los datos a la salida de esta.

En este caso, como datos de entrada, la función espera recibir, en forma de flujo de datos, una imagen en formato YUV 4:2:2. Para obtener la imagen, se ha hecho uso de las herramientas de OpenCV que permiten la lectura, tratamiento y almacenamiento de

imágenes de forma sencilla. El paquete Vivado HLS ya tiene incluida las funciones básicas de OpenCV para ser empleadas en un *testbench*. Para poder utilizarlas, únicamente es necesario añadir a este fichero la directiva *#include <hls_opencv.h>*. Cabe destacar que estas funciones no son sintetizables, por lo que únicamente pueden ser utilizadas en el banco de pruebas.

Sin embargo, una vez leída la imagen, esta se encuentra en formato *Mat*, propio de OpenCV. Para convertir la imagen a un flujo de datos AXI se ha utilizado la función *cvMat2AXIvideo* desarrollada por Xilinx y disponible también en la biblioteca *hls_opencv*.

Una vez se dispone de los datos de entrada en el formato YUV4:2:2, se llama a la función *top*, pasando como argumento la imagen de entrada. Finalizada la ejecución de esta función, se obtiene un flujo de datos de tipo *AXI-Stream* que, tras la conversión al formato *Mat*, se almacena en forma de imagen.

Sin embargo, como se ha comentado anteriormente, puesto que se necesita una imagen entera antes de disponer del histograma asociado, este se muestra en el siguiente *frame*. Para simular esto, se repite el proceso descrito anteriormente, llamando en el programa de *testbench* de nuevo a la función *top*.

De esta forma, la imagen obtenida por esta segunda ejecución será la que contenga superpuesto el histograma obtenido de la primera.

En la Figura 4.2 (izquierda), se muestra a modo de ejemplo la ejecución de la simulación del código C de HLS. Los resultados obtenidos se han comparado con los calculados para la misma imagen por la función *imhist* en Matlab. Esta función calcula el histograma de una imagen y lo representa por medio de barras cuya altura representa el número de píxeles para cada uno de los niveles de gris, tal y como se puede observar en la Figura 4.2 (derecha).

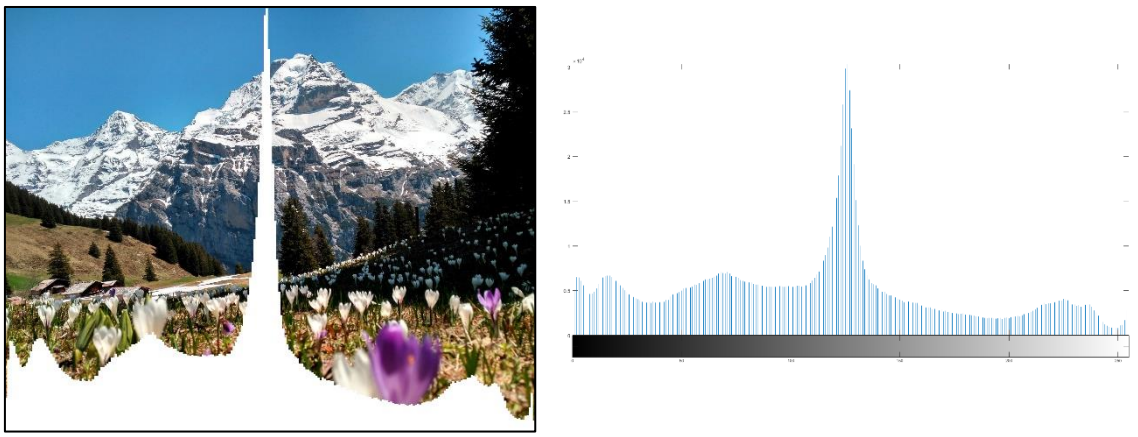


Figura 4.2: Imagen obtenida por la simulación C en HLS (izquierda) e histograma mostrado por la función imhist en Matlab

Con el objetivo de comparar de forma cuantitativa ambos resultados, se ha normalizado el histograma obtenido por la función de Matlab de la misma forma que se normaliza el histograma en el algoritmo implementado en HLS. Así, cada intervalo del histograma tendrá una altura entre 0 y 1023, representando la altura en píxeles que tendrá cada una de las barras en la imagen generada por el bloque *hardware*. Cabe destacar que estos cálculos se han realizado utilizando coma fija, con un tamaño de palabra y parte entera igual a la utilizada en HLS.

Por otro lado, a partir de la imagen obtenida por la simulación C en HLS, se debe conocer la altura de cada una de las barras que componen el histograma superpuesto en la imagen. Para ello, se ha separado este histograma y se ha calculado cuantos píxeles de altura tiene cada uno de los intervalos.

Tras el cálculo de los valores anteriores, estos se han comparado, siendo exactamente iguales.

4.2.4 Síntesis, cosimulación y exportación del diseño

Una vez que se ha comprobado el correcto funcionamiento de la función, se debe llevar a cabo el proceso de síntesis. Este proceso genera un informe en el cual se indica una estimación de la frecuencia máxima de funcionamiento, la latencia y los recursos utilizados. Cabe destacar que estos recursos son comparados con la totalidad de los disponibles en el dispositivo en el cual se implementará. De esta forma, el desarrollador puede conocer, de forma aproximada, si dispone de suficientes recursos para implementar en una determinada tarjeta el diseño realizado.

En la Figura 4.3 se puede observar los resultados obtenidos tras la síntesis de la función desarrollada en este capítulo.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	6.67	6.33	0.83

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2623239	2623239	1314820	1314820	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	0	-	20	80
Instance	1	2	6146	8321
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	6	-
Total	1	2	6172	8401
Available	280	220	106400	53200
Utilization (%)	~0	~0	5	15

Figura 4.3: Estimación obtenida tras el proceso de síntesis

Posteriormente, con el bloque ya sintetizado, se puede realizar la cosimulación C/RTL de manera que se tenga una simulación más fidedigna del comportamiento del bloque HW generado. De esta manera se simula la aplicación teniendo en cuenta la implementación RTL final, por lo que los resultados obtenidos son los esperables una vez se lleve a cabo la descarga real del sistema en una tarjeta de desarrollo.

En la Figura 4.4 se puede observar la ventana de configuración de la cosimulación C/RTL. En esta ventana se pueden seleccionar diferentes parámetros como el lenguaje de descripción *hardware* a emplear, VHDL o Verilog, o el simulador a utilizar.

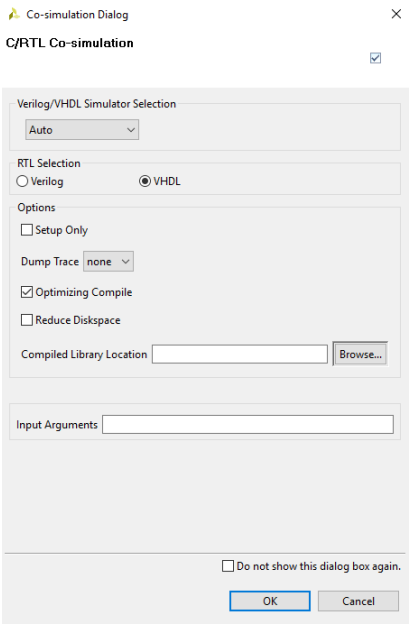


Figura 4.4: Cosimulación C/RTL

Tras la ejecución de la cosimulación, cuyo proceso se prolonga durante aproximadamente 20 minutos en el caso de esta función, se obtiene un archivo de salida con la imagen y el histograma superpuesto. Al igual que en el caso de la simulación C, el histograma se ha comparado con el obtenido en Matlab, obteniendo los mismos resultados.

Además, se obtiene un reporte, mostrado en la Figura 4.5, con dos apartados, *latency* y *interval*. La latencia indica los ciclos de reloj que tarda el algoritmo implementado en ejecutarse en *hardware* y el apartado *interval* refleja cuantos ciclos es necesario esperar hasta que el bloque pueda aceptar nuevos datos.

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	Pass	1317892	1318930	1319968	1317888	1318926	1319964
Verilog	NA	NA	NA	NA	NA	NA	NA

Figura 4.5: Resultados obtenidos tras la cosimulación del algoritmo

Se puede observar que los ciclos obtenidos son ligeramente superiores al número de píxeles que componen un *frame* por lo que, suponiendo que existen ciertos ciclos de reloj entre dos imágenes generadas consecutivamente por el sensor PYTHON, se podrán procesar todos los *frames* capturados por la cámara.

Como los resultados son los esperados, el siguiente paso es proceder a exportar el bloque IP de forma que se genere toda la información relacionada con el bloque y éste pueda ser añadido a un repositorio de Vivado. En el proceso de exportación, mostrado en la Figura 4.6, se puede indicar el nombre del diseño y la versión de este, entre otros parámetros.

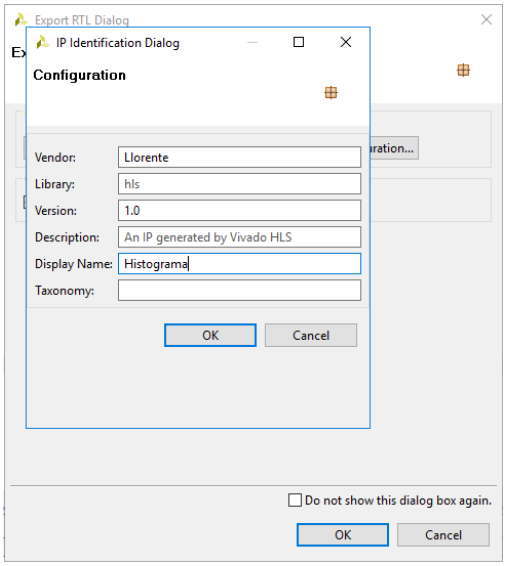


Figura 4.6: Exportación bloque IP

4.3 Adición del módulo IP al diseño en Vivado

Una vez que se ha generado el bloque IP en un repositorio de bloques que puede ser local al proyecto, se debe añadir al diseño completo explicado en el capítulo anterior. Para ello, el primer paso es añadirlo al conjunto de bloques seleccionables en el entorno Vivado indicando la localización del proyecto HLS en la pestaña *Project Settings -> IP -> Repository Manager*. A partir de este momento el bloque HLS se encontrará disponible en el apartado *Windows -> IP Catalog* y puede ser añadido al diseño, mostrando un aspecto como el que se puede observar en la Figura 4.7.

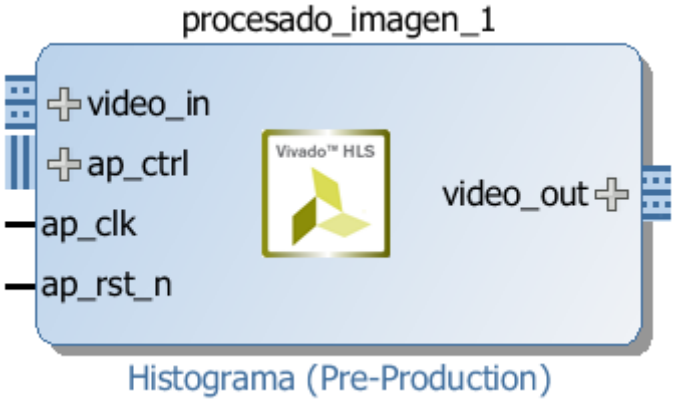


Figura 4.7: Bloque IP de la función implementada en HLS en Vivado

Este bloque debe ser añadido al diseño explicado en el capítulo anterior, ubicándose entre el bloque *Chroma Resampler* y *VDMA*. Las interfaces disponibles en este bloque se enumeran a continuación, indicando para cada una de ellas su utilidad y donde debe ser conectada:

- *video_in*: puerto *AXI-Stream* esclavo por el cual se recibe el flujo de píxeles en formato YUV 4:2:2. Debe ser conectado al puerto maestro del bloque *Chroma Resampler*.
- *video_out*: puerto *AXI-Stream* maestro por el cual se genera el flujo de píxeles en formato YUV 4:2:2 de la imagen con el histograma superpuesto. Este puerto es conectado a la interfaz esclava del bloque *VDMA*.

- *ap_clk*: señal de reloj del bloque HLS. Se conecta al reloj de 150 MHz, *FCLK_CLK1*.
- *ap_rst_n*: señal de *reset* del bloque HLS. Se conecta al bloque *Processor System Reset* asociado a la señal de reloj de 150 MHz.
- *ap_ctrl*: conjunto de señales que permiten el control del bloque HLS. En este caso, únicamente se conecta la señal *ap_start* a un nivel alto constante, de forma que el bloque se ejecute constantemente.

En la Figura 4.8 se puede observar el diseño completo en Vivado, una vez se ha añadido el bloque HLS al resto del sistema descrito en el capítulo anterior.

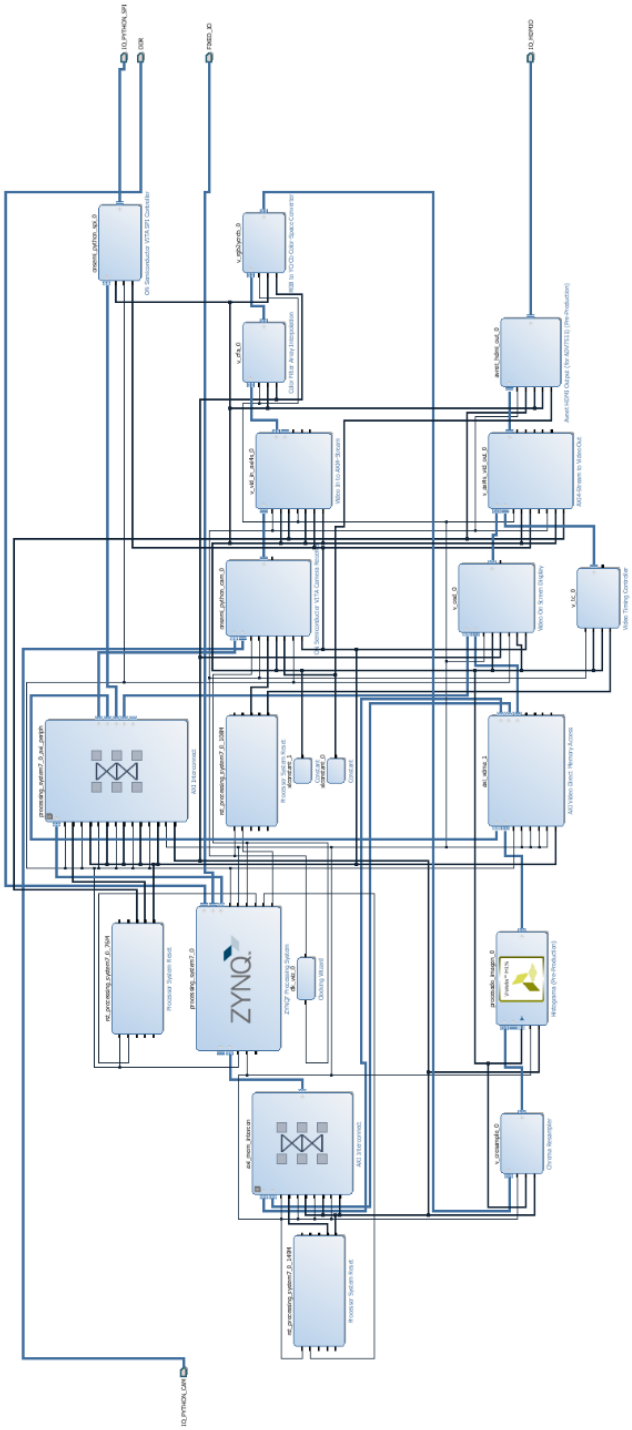


Figura 4.8: Diseño completo de la aplicación en Vivado para el cálculo del histograma

Una vez se han realizado todas las conexiones necesarias, se procede a realizar la síntesis, la implementación y la generación del archivo *bitstream* que será posteriormente utilizado para configurar el *hardware* programable. Este proceso tarda aproximadamente 20 minutos en completarse.

Cuando el proceso culmina, se genera un informe donde se recoge, entre otros parámetros, el uso de recursos, mostrados en la Figura 4.9, que hace la aplicación desarrollada en relación a los disponibles en el dispositivo utilizado. Se puede observar que estos son muy similares a los obtenidos antes de añadir el bloque del cálculo del histograma al proyecto.

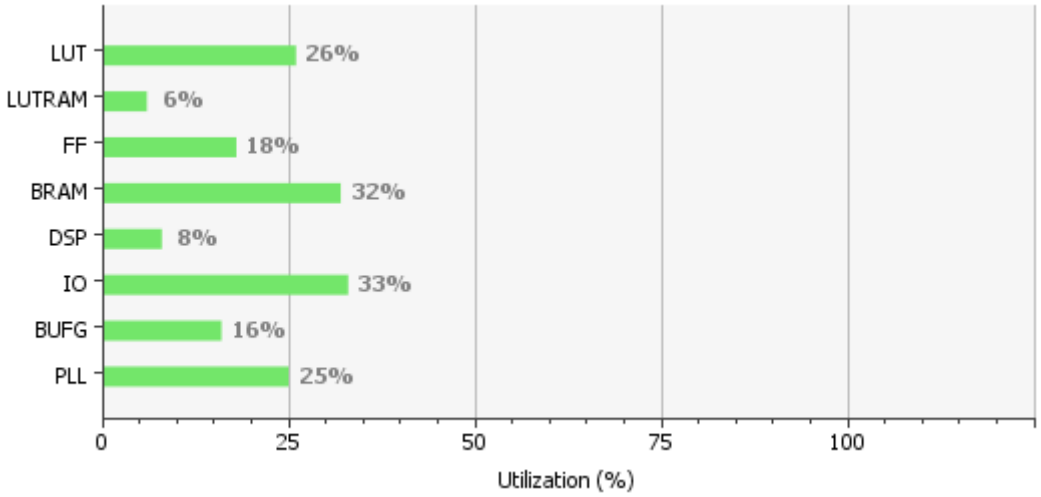


Figura 4.9: Recursos utilizados tras la implementación en Vivado

4.4 Vivado SDK - Ejecución de la aplicación en placa

El bloque IP diseñado en este capítulo no necesita ninguna configuración adicional a nivel *software*. Así, el código utilizado en esta aplicación es el mismo explicado en el capítulo anterior.

Tras la ejecución de la aplicación, en el monitor se pueden observar las imágenes capturadas por el sensor PYTHON y, superpuestas a estas, sus histogramas asociados. Un ejemplo del funcionamiento de esta aplicación se puede observar en la Figura 4.10.

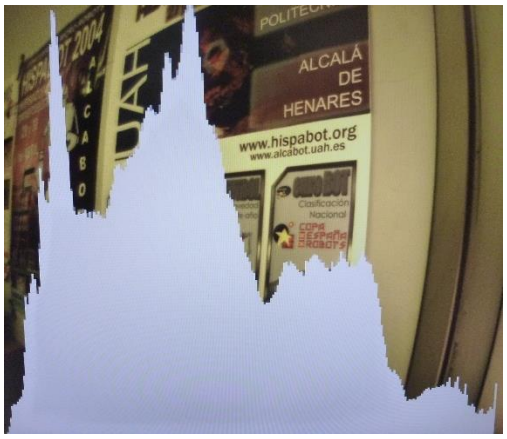


Figura 4.10: Imagen capturadas por el sensor PYTHON con su histograma asociado

Al igual que en el capítulo anterior, se ha calculado el número de imágenes por segundo escritas en los *buffers* utilizando las señales *user* de los buses *AXI-Stream*. El resultado

obtenido sigue siendo el mismo, aproximadamente 37 *frames* por segundo, lo que supone un procesamiento en tiempo real de las imágenes capturadas por el sensor.

Capítulo 5: Alternativas de implementación del algoritmo HOG

En los capítulos anteriores se ha desarrollado un sistema de captura de imágenes provenientes de un sensor de imagen y su posterior visualización en tiempo real a través de una interfaz HDMI. A continuación, se ha añadido un sencillo algoritmo de procesamiento *hardware* a estas imágenes con el objetivo de conocer el histograma de estas en tiempo real.

Sin embargo, un *SoC* como el dispositivo Zynq permite implementar algoritmos de procesamiento de vídeo más complejos y costosos computacionalmente que permitan, por ejemplo, obtener información de los objetos que se pueden observar en una determinada imagen. Uno de estos algoritmos es el histograma de gradiente orientado (HOG) cuya ejecución calcula un descriptor que permite conocer los contornos predominantes de una determinada imagen. Además, este algoritmo puede ser empleado conjuntamente con otros algoritmos como las máquinas de vector soporte (SVM) para identificar determinados elementos en una imagen como, por ejemplo, personas o caras.

A lo largo de este capítulo se explicará detalladamente el funcionamiento del algoritmo HOG, así como todos los parámetros que lo definen. Posteriormente, se comentarán todas las simplificaciones efectuadas para realizar un codiseño HW/SW. Por último, el módulo *hardware* implementado será utilizado en un programa *software* e implementado en la tarjeta de desarrollo descrita en capítulos anteriores.

5.1 Algoritmo HOG

El histograma de gradientes orientados (*Histogram of Oriented Gradients* o HOG) es un descriptor de características usado en procesamiento de imágenes para la detección de objetos. Este método consiste en contabilizar las ocurrencias de la orientación del gradiente en porciones de la imagen denominadas ventanas de detección o regiones de interés (ROI).

El resultado obtenido tras la aplicación del algoritmo es un vector con los histogramas calculados en cada ventana de detección denominado *descriptor HOG*. Cada histograma del descriptor agrupa las diferentes variaciones del gradiente en pequeñas regiones de la ventana de detección, permitiendo conocer las direcciones predominantes y, por tanto, los contornos de los objetos. Esta información puede ser utilizada posteriormente para determinar la existencia de un determinado objeto en la ventana analizada.

Para extender el procesamiento a una imagen completa, es necesario desplazar la ventana de detección por esta, obteniendo para cada desplazamiento un descriptor HOG independiente. Así, los resultados tras el procesamiento de una imagen serán un conjunto de descriptores HOG independientes entre sí y cuyo número dependerá de la superposición de las diferentes ventanas procesadas.

5.1.1 Definición de parámetros del algoritmo HOG

Para poder comprender el funcionamiento del algoritmo HOG, es necesario definir primero una serie de parámetros que lo caracterizan:

- Ventana de detección o región de interés (ROI): sección de una imagen que describirá un descriptor HOG.
- Celda: subconjunto de la ventana de detección que agrupa a un conjunto de píxeles adyacentes. Para cada celda se calcula un histograma a partir de los valores de gradiente de los píxeles que la conforman. Las celdas pueden tener diferentes

tamaños, pero, en general, tienen un formato cuadrado, es decir, agrupan el mismo número de píxeles por fila que por columna.

- Bloque: conjunto de celdas adyacentes utilizadas para la normalización de los histogramas de cada una de ellas. El número de celdas en cada dirección debe ser el mismo, manteniendo de esta forma un formato de bloque cuadrado.
- *Stride* de bloque: indica la superposición de las celdas en la definición de los bloques. Es decir, señala si dos bloques vecinos comparten alguna de las celdas entre ellos.

5.1.2 Procesamiento HOG

El algoritmo HOG fue descrito por primera vez por Robert K. McConnell en una patente en el año 1986 aunque hasta el año 1994 no fue denominado de esta forma por los laboratorios Mitsubishi Electric. Sin embargo, su uso no estuvo muy extendido hasta el año 2005 cuando Navneet Dalal y Bill Triggs del *National Institute for Research in Computer Science and Automation* (INRIA) presentaron en la *Conference on Computer Vision and Pattern Recognition* (CVPR) su trabajo *Histograms of Oriented Gradients for Human Detection* [12].

Los diferentes pasos para el cálculo del algoritmo HOG, explicados en los apartados siguientes, han sido extraídos de la tesis *Finding People in Images and Videos* publicada por Navneet Dalal en 2006 [13].

5.1.2.1 Normalización Gamma/Color

Este primer paso del proceso consiste ecualizar la imagen de entrada con el objetivo de reducir la influencia de los efectos de iluminación. Para ello, se puede aplicar una corrección gamma de compresión que ayuda a reducir los efectos de sombras y variaciones de iluminación. Si la imagen de entrada es en color, esta corrección es aplicada a cada uno de los canales que la conforman.

5.1.2.2 Cálculo gradiente

El segundo paso consiste en el cálculo del gradiente para cada uno de los píxeles que componen la imagen. El gradiente da información de las variaciones de luminosidad entre píxeles vecinos lo que permite conocer los contornos de los objetos que se encuentran en la imagen. En el caso de imágenes en color, el gradiente debe ser calculado separadamente para cada uno de los canales y elegir aquel gradiente que tenga una mayor norma como el gradiente del píxel analizado.

En el caso de imágenes en escala de grises, para cada píxel se calcula su gradiente, es decir, la variación del valor de luminancia con los píxeles vecinos, tanto en sentido horizontal como en sentido vertical. Para ello, se aplican dos máscaras, mostradas en la Figura 5.1, una para cada dirección.

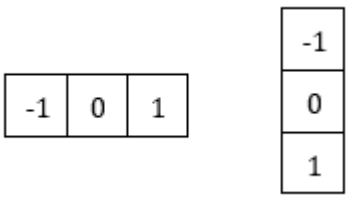


Figura 5.1: Máscara horizontal (izquierda) y máscara vertical (derecha)

La primera de las máscaras, Figura 5.1 (izquierda), detecta cambios del valor de luminancia en dirección horizontal (líneas verticales) y la segunda de las máscaras, Figura 5.1 (derecha), en dirección vertical (líneas horizontales).

Las dos máscaras se aplican por separado, obteniéndose dos valores, *Gx* y *Gy*. El valor *Gx* indica la variación de la luminancia y su sentido en dirección horizontal y *Gy* en dirección vertical.

En la Figura 5.2 se muestran los gradientes obtenidos para diferentes ejemplos. En el ejemplo 1, se observa una discontinuidad vertical entre una zona oscura con una zona clara. Aplicando la máscara horizontal en alguno de los píxeles que se encuentren en el límite de una de las zonas, se obtiene un valor elevado debido al gran contraste entre ambas. *Gx* será además positivo, según se ha configurado la máscara en la Figura 5.1 (izquierda), ya que la discontinuidad se produce de una zona oscura a una zona clara. Aplicando la máscara vertical, se puede observar que el resultado de *Gy* es 0 ya que los píxeles vecinos superior e inferior serán iguales.

En el ejemplo 2, se observa una discontinuidad horizontal entre una zona clara y una zona oscura. Al contrario que en el caso anterior, la máscara horizontal será 0 y la vertical tendrá un valor elevado y negativo, según se ha configurado la máscara en la Figura 5.1 (derecha).

El ejemplo 3 es muy similar al 1 pero el contraste entre las dos zonas es menor. Por ello, el valor de *Gx* es menor que el obtenido en el caso 1.

Por último, el ejemplo 4 muestra una situación en la que tanto *Gx* como *Gy* toman valores distintos de 0.

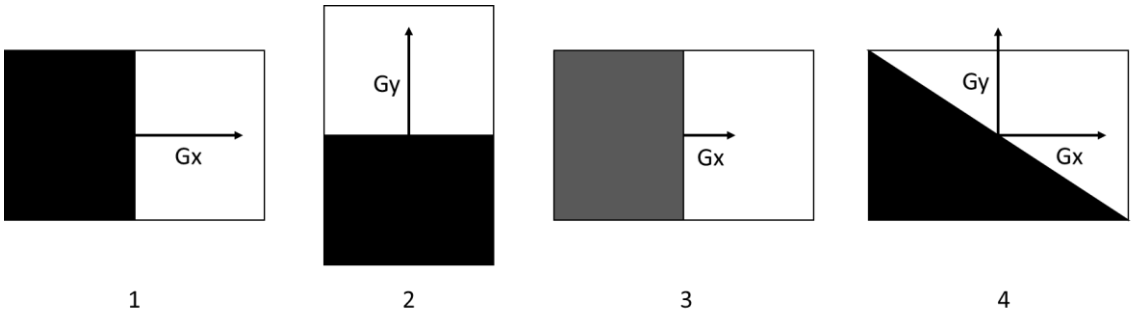


Figura 5.2: Ejemplos cálculo gradientes horizontal y vertical

Los valores *Gx* y *Gy* pueden ser combinados para conformar un vector gradiente para cada píxel. Este vector gradiente señalará la dirección de la máxima variación de luminancia y su valor, tal y como puede verse en la Figura 5.3. La magnitud y orientación de dicho vector puede calcularse utilizando las siguientes fórmulas:

$$Magnitud = \sqrt{(Gx)^2 + (Gy)^2}$$

$$Orientación = \tan^{-1}\frac{-Gy}{Gx}$$

La magnitud indica cual es la intensidad del gradiente teniendo en cuenta las dos componentes estudiadas. Se denomina también módulo del vector y se calcula mediante la norma.

La orientación del vector nos indica la dirección del gradiente, medida como el ángulo que forma el vector con el eje horizontal. El signo menos del numerador es debido a como se ha configurado la máscara vertical y al sentido de las coordenadas del eje vertical.

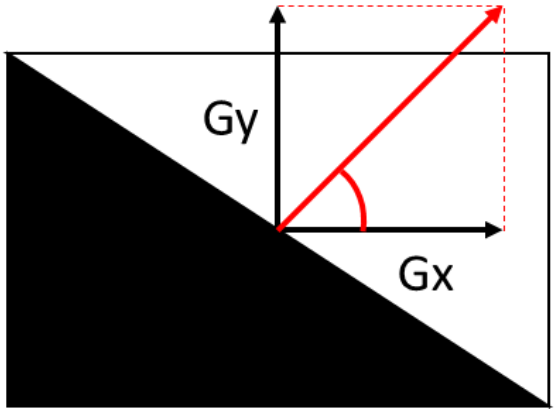


Figura 5.3: Magnitud y orientación del gradiente

5.1.2.3 Generación de histogramas

A partir de los gradientes calculados en el apartado anterior y agrupando los píxeles por regiones denominadas *celdas*, se van a obtener los histogramas que reflejarán las direcciones de máxima variación de la luminosidad.

De esta forma, cada píxel contribuye a la formación del histograma asociado a la celda a la que pertenece votando al intervalo de orientación correspondiente a su gradiente. Los intervalos de orientación deben estar uniformemente espaciados entre 0 y 360 grados si el sentido es importante (se quiere poder diferenciar entre transiciones de zonas claras a zonas oscuras y viceversa) o entre 0 y 180 grados si el sentido del gradiente es irrelevante. En este último caso, un gradiente negativo contribuye al mismo intervalo al que contribuiría su valor absoluto. El número de intervalos de cada histograma permite ajustar la precisión en la clasificación de los gradientes.

El cálculo de los histogramas a partir de la orientación del gradiente supone que dos gradientes con orientaciones muy similares pueden ser asignados a intervalos diferentes. Esto supone que el histograma calculado para una determinada celda sea muy sensible a las pequeñas variaciones del gradiente. Para evitar esto, se realiza una interpolación en orientación de forma que un mismo gradiente puede afectar a dos intervalos vecinos del histograma. De esta forma, la orientación del gradiente determina entre que dos intervalos vecinos se reparte proporcionalmente su módulo. Así, dependiendo de cómo de centrado en un intervalo esté la orientación del gradiente, este afectará más o menos a ese determinado intervalo y a su vecino. A modo de ejemplo, si un gradiente toma el valor exacto correspondiente al punto central de un intervalo, este será el único intervalo afectado por el píxel estudiado. Sin embargo, si el hecho anterior no se cumple, el módulo del gradiente afectará a los dos intervalos cuyos centros se encuentren más próximos a la orientación del gradiente. Así, cuanto más cerca se encuentre del valor central de un intervalo más afectará a dicho intervalo y menos a su vecino más cercano. En el caso de que la orientación del gradiente sea exactamente el valor frontera entre dos intervalos, el módulo del gradiente será repartido de forma equitativa entre los dos intervalos.

De forma similar, la división en celdas de una ventana de detección supone que píxeles muy cercanos entre sí son asignados a celdas diferentes y, por tanto, influyen en histogramas diferentes. Esto supone que pequeñas variaciones en la forma de un objeto modifiquen los histogramas en gran medida. Para evitar esto, se emplea la interpolación espacial de forma que cada gradiente es repartido de forma proporcional a la distancia del píxel al centro de cada una de las cuatro celdas más cercanas.

La aplicación de estas dos interpolaciones se combina mediante el empleo de una interpolación trilineal, explicada detalladamente en [13].

5.1.2.4 Normalización del histograma

La intensidad de los gradientes se ve muy afectada por las variaciones locales en la iluminación y los diferentes contrastes entre el primer plano y el fondo. Con el objetivo de reducir la influencia del contraste es necesario normalizar los histogramas generados.

La técnica más utilizada consiste en agrupar las celdas en grupos más grandes denominados *bloques* y utilizar los gradientes de todos los píxeles asociados para normalizar los histogramas de cada celda. El tamaño de bloque determina la capacidad de suprimir los cambios de iluminación local. De esta forma, un tamaño de bloque reducido permite suprimir mejor los cambios de iluminación local.

Diferentes normas pueden ser aplicadas, siendo las más habituales las siguientes:

- *L2-norm*: $v = \frac{v}{\|v\|_2^2+\epsilon^2}$ donde $\|v\|_2^2 = \sqrt{\sum_{i=1}^{n}|v_i|^2}$ y ϵ una constante para evitar dividir entre 0.
- *L2-hys*: Primero se realiza una normalización L2 y posteriormente se recortan todos los valores mayores de 0,2 a dicho valor. Finalmente, se aplica otra normalización L2.
- *L1-norm*: $v = \frac{v}{\|v\|_1+\epsilon}$ donde $\|v\|_1 = \sum_{i=1}^{n}|v_i|$ y ϵ una constante para evitar dividir entre 0.
- *L1-sqrt*: $v = \sqrt{\frac{v}{\|v\|_1+\epsilon}}$ donde $\|v\|_1 = \sum_{i=1}^{n}|v_i|$ y ϵ una constante para evitar dividir entre 0.

Con el objetivo de detectar mejor los cambios de iluminación local, los bloques pueden superponerse entre sí, de forma que una misma celda puede pertenecer a varios bloques vecinos. Esto permite obtener más información ya que un mismo histograma es normalizado varias veces, cada una de ellas con celdas vecinas diferentes.

5.1.2.5 Vector de características HOG

El descriptor HOG es un vector formado por la concatenación de cada uno de los histogramas normalizados que componen una ventana de detección. El tamaño de un descriptor HOG vendrá determinado por los parámetros que definen el algoritmo.

Por un lado, el tamaño de la ventana de detección y el tamaño de celda determinan el número de histogramas a calcular. Por otro lado, el tamaño de cada histograma está determinado por el número de intervalos de clasificación. Por último, el tamaño y el *stride* de bloque determina el número de veces que es normalizado el histograma de una celda. Así, el tamaño de descriptor HOG vendrá determinado por:

$$Tamaño\ descriptor = \frac{Nº\ bloques\ en\ horizontal}{ventana} * \frac{Nº\ bloques\ en\ vertical}{ventana} * \frac{Nº\ celdas}{bloque} * \frac{Nº\ de\ intervalos}{histograma}$$

El orden de aparición de los histogramas en el descriptor HOG se agrupa por bloques siguiendo primero la dirección vertical y, posteriormente, la dirección horizontal.

5.1.3 Parámetros HOG para detección de personas

Como se ha comentado anteriormente, el algoritmo HOG dispone de varios parámetros que permiten adaptar sus capacidades dependiendo de los objetos que se deseen detectar. En [13] se realiza un estudio de todos estos parámetros enfocado a la detección de personas utilizando una máquina de vector soporte. Para ello, se entrena una SVM con los descriptores HOG obtenidos para los diferentes parámetros HOG y se testea el algoritmo. A partir de los aciertos y errores en la búsqueda de personas, se seleccionan cuáles son los mejores parámetros para este cometido.

De esta forma, se determina que los parámetros que mejor resultado logran en la detección de personas son los siguientes:

- Tamaño de ventana: 64 píxeles de ancho por 128 de alto. Es el tamaño que suele ser utilizado por defecto en la detección de personas.
- Tamaño celda: Celdas de 6x6 píxeles y 8x8 píxeles logran los mejores resultados.
- Intervalos por histograma: 9 intervalos por histograma sin importar el sentido de la orientación. De esta forma, cada intervalo agrupa 20 grados de inclinación. Resultados muy similares empleando 18 intervalos por histograma utilizando el signo del gradiente.
- Tamaño bloque: Bloques de 3x3 celdas y 2x2 celdas consiguen los menores índices de error.
- *Stride* bloque: Los mejores resultados se obtienen superponiendo los bloques entre sí, de forma que cada celda pertenece a varios bloques. El mejor resultado se obtiene para la superposición de la mitad de cada bloque vecino.

En la Figura 5.4 se puede observar gráficamente una ventana de detección dividida en celdas de 8x8 píxeles y en bloques de 2x2 celdas.

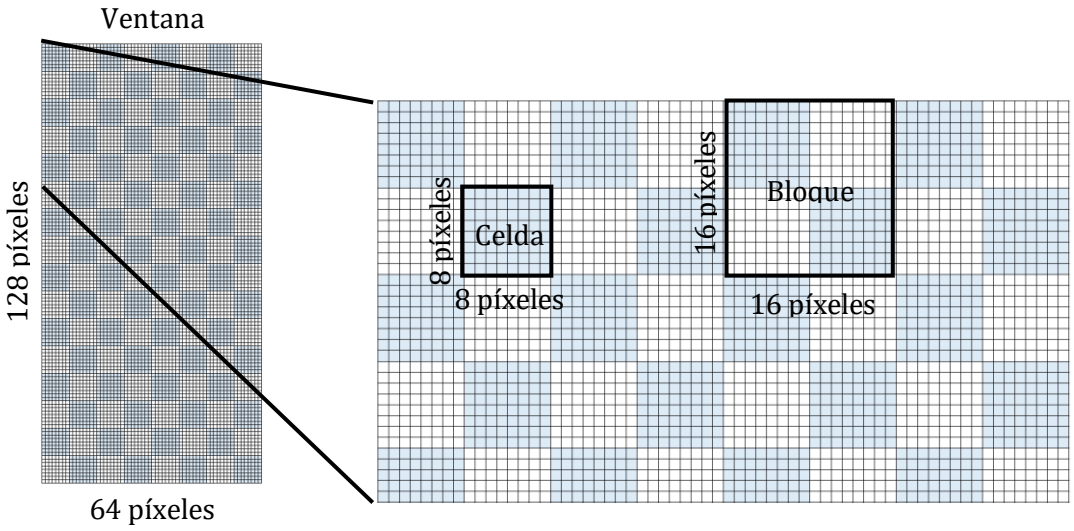


Figura 5.4: Parámetros HOG para detección de personas

En la Figura 5.5 se puede observar, a modo de ejemplo, el desplazamiento de los bloques a través de la ventana de detección con un *stride* de 8 píxeles por dirección.

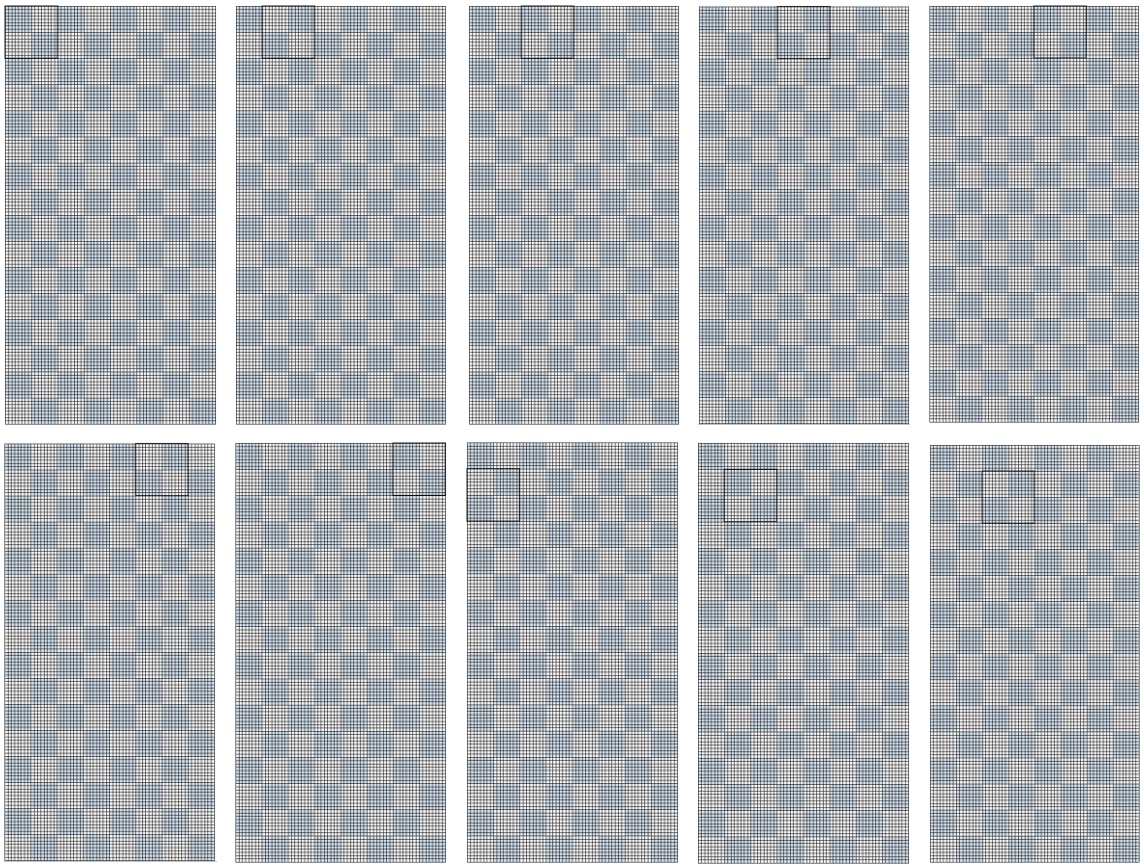


Figura 5.5: Stride de bloque de 8 píxeles

5.1.4 Función HOG de Matlab

El algoritmo HOG está implementado en Matlab a través de la función *extractHOGFeatures*. Esta función está incluida en la *toolbox Computer Vision System* y está programada siguiendo el procedimiento descrito por Dalal y explicado en el apartado anterior.

Cabe destacar que, antes del cálculo de los histogramas se aplica un filtro paso bajo gaussiano con una desviación estándar igual a la mitad de la anchura de bloque a los módulos de los gradientes con el objetivo de reducir la influencia de los píxeles más cercanos a los bordes de los bloques.

Además, el método de normalización utilizado es el denominado *L2-Hys* que se basa en aplicar primero una normalización *L2-norm*, posteriormente una umbralización de 0,2 y, por último, otra normalización *L2-norm*.

La función recibe como parámetro principal la imagen que desea ser procesada. Esta imagen puede ser en color (3 canales) o en escala de grises. Además, de forma opcional, se puede indicar el valor de los distintos parámetros característicos del descriptor HOG:

- *CellSize*: Se emplea para fijar el número de píxeles que conforman una celda. Se especifica como un vector de dos elementos, cada uno de ellos indica el número de píxeles en cada lado de la celda. El valor por defecto especifica un tamaño de celda de 8x8 píxeles.
- *BlockSize*: Configura el número de celdas que conforman un bloque. Se indica con un vector de dos elementos, cada uno referido al número de celdas en cada dirección. El valor por defecto especifica un bloque conformado por 2x2 celdas.

- *BlockOverlap*: Refleja el número de celdas superpuestas entre dos bloques adyacentes. Se especifica como un vector de 2 elementos, cada uno de ellos configurando una dirección. El valor por defecto es $BlockSize/2$ lo que supone una superposición de una celda entre bloques contiguos.
- *NumBins*: Número de intervalos en cada histograma. El valor por defecto es 9 intervalos.
- *UseSignedOrientation*: Permite seleccionar el empleo o no del signo de los gradientes. El valor por defecto es *false* por lo que el sentido no es tenido en cuenta.

Como se ha comentado anteriormente, al producirse solapamiento de bloques, las celdas que no se encuentran en los bordes de la ventana de detección pertenecen a varios bloques. Con la parametrización fijada anteriormente, los histogramas asociados a estas celdas son normalizados 4 veces. Las celdas que se encuentran en el borde de la ROI y no pertenecen a las esquinas de esta pertenecen a dos bloques y, únicamente las celdas de las esquinas pertenecen a un único bloque. Esto puede ser observado en la Figura 5.6, donde se representan todas las celdas que componen una ventana de detección y el número de veces que sus histogramas asociados son normalizados.

1	2	2	2	2	2	2	1
2	4	4	4	4	4	4	2
2	4	4	4	4	4	4	2
2	4	4	4	4	4	4	2
2	4	4	4	4	4	4	2
2	4	4	4	4	4	4	2
2	4	4	4	4	4	4	2
2	4	4	4	4	4	4	2
2	4	4	4	4	4	4	2
2	4	4	4	4	4	4	2
2	4	4	4	4	4	4	2
2	4	4	4	4	4	4	2
2	4	4	4	4	4	4	2
2	4	4	4	4	4	4	2
2	4	4	4	4	4	4	2
1	2	2	2	2	2	2	1

Figura 5.6: Histogramas normalizados por celda

La función devuelve como salida el descriptor HOG en un vector fila. El tamaño del vector para una imagen de entrada de 64x128 píxeles y los parámetros característicos por defecto será:

$$\begin{aligned} &Tamaño\ descriptor \\ &\quad = 15\ bloques\ dirección\ vertical * 7\ bloques\ dirección\ horizontal \\ &\quad * 4\ celdas\ por\ bloque * 9\ intervalos\ por\ celda = 3780\ elementos \end{aligned}$$

Los elementos son ordenados por bloques, siguiendo primero la dirección vertical y, posteriormente la horizontal. Por cada bloque, cuatro histogramas son devueltos, uno para cada una de las celdas, colocados estos, también, siguiendo primero la dirección vertical y posteriormente la horizontal. Esta disposición puede ser observada en la Figura 5.7.

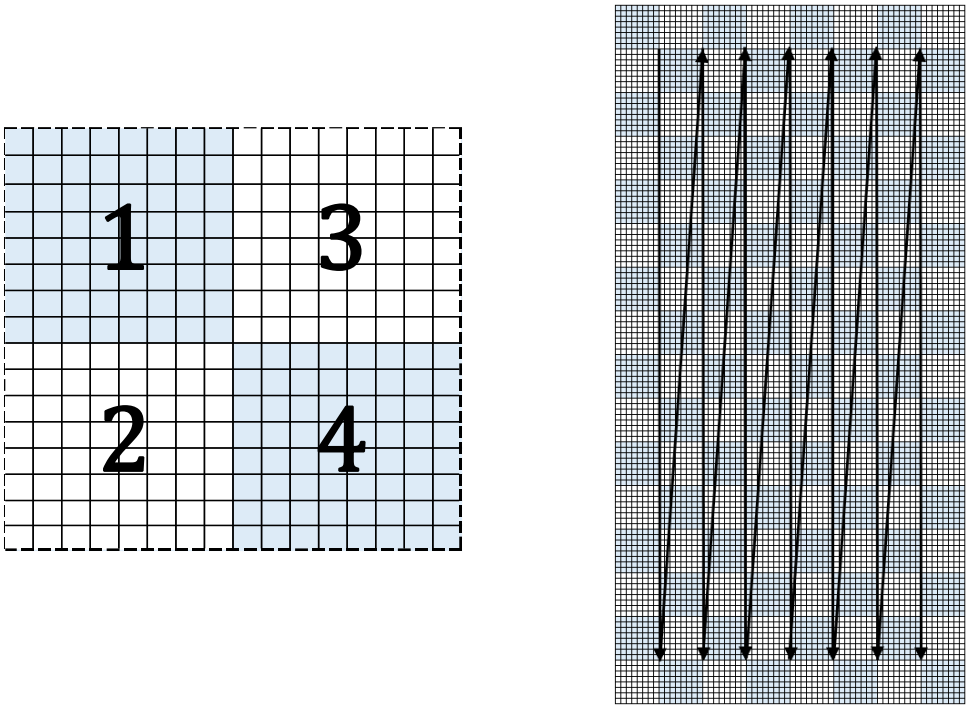


Figura 5.7: Orden de las celdas en un bloque (izquierda) y orden de los bloques en ROI (derecha)

Además, la función devuelve un objeto con la visualización del descriptor HOG obtenido. Un descriptor HOG es visualizado mostrando, para cada celda de la ventana de detección, los intervalos del histograma asociado. Así, para cada intervalo se dibuja una línea con centro el punto central de la celda y con el ángulo igual al valor medio del intervalo. La longitud de la línea será proporcional al valor del histograma para ese intervalo.

Cabe destacar que, en el caso de que el algoritmo HOG se haya configurado con solapamiento de bloques y, por tanto, cada celda aporte varios histogramas al descriptor, es necesario realizar una media de estos histogramas. Este histograma promediado será el representado en la celda correspondiente.

En el caso de haber configurado el algoritmo para no hacer uso del signo del gradiente el histograma estará comprendido entre 0 y 180 grados. Puesto que en ese caso el sentido del gradiente no ha sido tenido en cuenta, únicamente la dirección es representativa. Por ello, para cada intervalo representado es posible extender la línea en sentido contrario pasando por el centro de la celda.

El algoritmo HOG emplea las direcciones del gradiente entre píxeles para el cálculo de las orientaciones predominantes en las celdas que estos conforman. Las direcciones del gradiente son direcciones normales a los bordes que señalan. Por ello, existen dos formas habituales de representar los histogramas normalizados: reflejando las direcciones tal y como son obtenidas por el gradiente o representando la dirección normal a esta.

La función de Matlab representa las direcciones normales a las calculadas por el gradiente. De esta forma, las líneas representadas son paralelas a los bordes que describen permitiendo así comprobar visualmente los bordes detectados.

Un ejemplo de la representación de un descriptor HOG obtenida por la función *extractHOGFeatures* puede ser observada en la Figura 5.8.

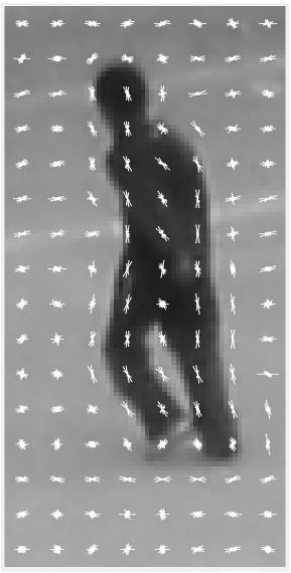


Figura 5.8: Representación descriptor HOG de la función de Matlab extractHOGFeatures

5.2 Implementación en SW del algoritmo HOG

Una vez que se ha explicado el algoritmo HOG y su funcionamiento con la función proporcionada desde Matlab, a continuación se describe el *pseudocódigo* C necesario para ejecutar el algoritmo HOG en SW, por ejemplo, en el procesador ARM del SoC Zynq.

```
void hog (unsigned char ventana[64x128], float valores_hog[1152])
{
      //Declaración de constantes: pesos filtro gaussiano e
interpolación trilineal
      for(y=0; y<128; y++)    //Se recorren todas las líneas de la
ventana de procesamiento
      {
            for(x=0; x<64; x++) //Se recorren todos los píxeles por
cada línea de la ventana
            {
                  //Cálculo del gradiente en dirección horizontal y
vertical: Gx y Gy
                  //Cálculo de la magnitud del gradiente:
mag=sqrt(Gx^2+Gy^2)
                  //Cálculo de la orientación del gradiente:
orientación=arctg(-Gy/Gx)
                  //Cálculo del intervalo del histograma asociado a la
orientación del gradiente
            }
      }

      for(j=0; j<4; j++)      //Para cada bloque en dirección
horizontal de la ventana de detección
      {
            for(k=0; k<8; k++)//Para cada bloque en dirección vertical
de la ventana de detección
            {
                  //Selección de los valores de magnitud y orientación
del bloque
                  //Aplicación de los pesos gaussianos a las magnitudes
de los gradientes
```

```
                    for(r=0; r<16; r++) //Para cada fila de un bloque
                    {
                          for(s=0; s<16; s++) //Para cada columna de un
bloque
                          {
                                //Interpolación trilineal
                          }
                    }
                    //Aplicación de la normalización L2
                    //Recorte de los valores superiores a 0,2
                    //Reaplicación de la normalización L2
              }
        }
}
```

Como se puede observar en el *pseudocódigo* anterior, el primer paso es calcular los gradientes de todos los píxeles que conforman una ventana de detección. Para ello, primero se calculan los gradientes en dirección horizontal y vertical y, a partir de la combinación de estos, se calcula la magnitud y la orientación del vector gradiente asociado. La magnitud se calcula como la norma L2 de los gradientes direccionales y la orientación como la arcotangente de la división de estos.

A partir de la orientación obtenida, se calcula a que intervalo del histograma se encuentra asociada esta. Además, se calcula como de centrado se encuentra la orientación del gradiente con el punto medio del intervalo asociado, permitiendo posteriormente ponderar la magnitud del gradiente con el intervalo vecino más cercano.

Para cada bloque que conforma una ventana de procesamiento, se aplica un filtro gaussiano a las magnitudes de los gradientes de forma que los píxeles ubicados en los extremos de los bloques tienen una menor influencia en el histograma que los píxeles ubicados en la parte central.

La generación de los histogramas se implementa mediante la interpolación trilineal que, además de la orientación del gradiente tiene en cuenta la posición de cada píxel en el bloque, tal y como se comentó en el apartado 5.1.2.3.

Por último, los histogramas calculados para un determinado bloque son normalizados empleando una norma L2. Una vez aplicada esta, se comprueban los valores obtenidos y se recortan aquellos que superen un umbral de 0,2 a dicho valor. Por último, se vuelve a aplicar la norma L2, obteniéndose finalmente el segmento del descriptor asociado al bloque procesado.

El algoritmo implementado en lenguaje C ha sido compilado, cargado y ejecutado en el procesador ARM del dispositivo Zynq y se han obtenido unos tiempos de procesamiento muy elevados, de en torno a 1 segundo por ventana de detección. Este tiempo de procesamiento es muy elevado y hace inviable su uso en cualquier aplicación de visión que posteriormente pueda desarrollarse empleando el dispositivo Zynq.

Por ello, en los siguientes apartados se explican detalladamente las diferentes pruebas, simplificaciones e implementaciones realizadas y cuyo objetivo es reducir drásticamente este tiempo de ejecución.

5.3 Implementación en HW del algoritmo HOG

A partir del código ejecutado en el procesador ARM, se ha intentado implementar el algoritmo a nivel *hardware* con el objetivo de acelerar su ejecución. Para ello, se ha utilizado la herramienta Vivado HLS que permite implementar a nivel *hardware* un algoritmo descrito en un lenguaje de alto nivel como C.

Sin embargo, tras realizar una primera aproximación se ha determinado que los recursos necesarios son muy elevados, tal y como se muestra en la Figura 5.9. En esta estimación, obtenida tras el proceso de síntesis, se puede observar que los recursos del dispositivo Zynq de la tarjeta de desarrollo MicroZed no son suficientes o suponen un alto porcentaje del total disponible. Además, cabe destacar la elevada latencia estimada para el algoritmo, lo que supondría un largo tiempo de ejecución.

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	6.67	8.10	0.83

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
3006331	6756987	3006332	6756988	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	16	0	7742
FIFO	-	-	-	-
Instance	6	85	26255	36813
Memory	134	-	2	1
Multiplexer	-	-	-	36919
Register	-	-	73641	-
Total	140	101	99898	81475
Available	280	220	106400	53200
Utilization (%)	50	45	93	153

Figura 5.9: Estimación temporal y de empleo de recursos tras síntesis del algoritmo HOG

El algoritmo HOG implementado tal y como se ha explicado en los apartados anteriores implica una gran cantidad de operaciones matemáticas de alto coste computacional. A continuación se enumeran, a modo de ejemplo, estas operaciones para alguno de los pasos en el cálculo de un descriptor HOG:

- Cálculo de la magnitud del gradiente: 2 multiplicaciones y una raíz cuadrada por cada píxel de la ventana de procesamiento.
- Cálculo de la orientación del gradiente: una división y el cálculo de una arcotangente por cada píxel de la ventana de procesamiento.
- Generación de los histogramas: la aplicación de la interpolación trilineal y el filtro gaussiano supone una gran cantidad de multiplicaciones por cada uno de los bloques que conforman una ventana de detección.

- Normalización de los histogramas: el empleo de la norma L2 supone, para cada bloque, 36 multiplicaciones y una raíz cuadrada, además de las 36 divisiones de los intervalos que componen los 4 histogramas de un bloque.

Como se puede observar en la enumeración anterior, una gran parte del algoritmo HOG emplea operaciones con un alto coste computacional que implican un gran uso de recursos *hardware* y una gran cantidad de ciclos para su cálculo. Por ello, se ha decidido modificar el algoritmo original, simplificándolo de forma que pueda ser ejecutado de forma más rápida, pero asegurándonos a su vez de que sigue siendo válido como forma de describir una imagen.

5.4 Simplificación del algoritmo HOG

El algoritmo HOG que se ha explicado en los apartados anteriores emplea operaciones como la raíz cuadrada, la función arcotangente, y la división para el cálculo del descriptor. Estas operaciones tienen un alto coste computacional lo que implica una gran cantidad de recursos cuando se implementan en *hardware* en un sistema reconfigurable.

Debido a la escasez de dichos recursos, se han realizado una serie de simplificaciones en el algoritmo que buscan obtener los resultados más similares posibles a los obtenidos con el algoritmo descrito anteriormente, pero utilizando operaciones más sencillas e implementables en *hardware*.

5.4.1 Parámetros HOG utilizados

Como se explicará en posteriores capítulos, el algoritmo HOG será empleado para la detección de personas en las imágenes capturadas por la cámara. Por ello, se ha utilizado una parametrización HOG similar a la indicada en el apartado 5.1.3. Así, se ha elegido una ventana detección de 128 píxeles de alto por 64 píxeles de ancho, dividida en celdas de 8x8 píxeles. Las celdas son agrupadas en bloques de 2x2 celdas y cada histograma estará formado por 9 intervalos, sin importar el sentido de la orientación del gradiente.

Sin embargo, con el objetivo de disminuir el número de operaciones a realizar, se ha elegido un *stride* de bloque igual al tamaño de este. De esta forma, cada celda es normalizada una única vez ya que únicamente pertenece a un bloque. Así, la ventana de detección estará conformada por 8 bloques en dirección vertical y 4 bloques en dirección horizontal.

Cabe destacar que el empleo de un *stride* igual al tamaño de bloque, aunque obtiene resultados de detección en personas ligeramente inferiores si se comparan con el empleo de un *stride* de bloque menor [13], sigue siendo completamente válido para dicha función, tal y como posteriormente se comentará en este TFM.

La configuración de los distintos parámetros puede ser observada en la Figura 5.10. Cabe destacar que todos los bloques que componen la ventana de detección han sido destacados con línea discontinua.

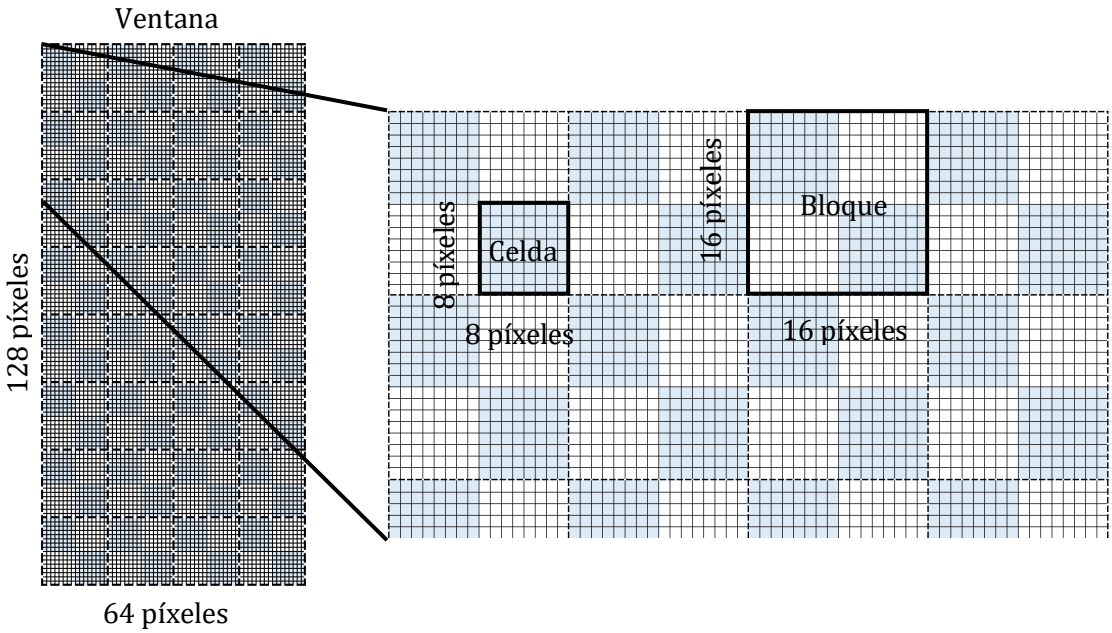


Figura 5.10: Configuración HOG simplificado

La configuración anterior supone que el descriptor HOG disminuya su tamaño hasta estar compuesto por 1152 elementos:

$$Tamaño\ descriptor = 8\ bloques\ dirección\ vertical * 4\ bloques\ dirección\ horizontal * 4\ celdas\ por\ bloque * 9\ intervalos\ por\ celda = 1152\ elementos$$

5.4.2 Cálculo magnitud del gradiente

El cálculo del módulo, según se ha explicado en el apartado 5.1.2.2, supone el empleo de la operación norma y, por tanto, el cálculo de dos potencias, una suma y una raíz cuadrada. Las potencias y la raíz cuadrada son operaciones que suponen una gran cantidad de recursos en una implementación *hardware*.

Los gradientes en dirección horizontal y vertical, *Gx* y *Gy*, calculados tal y como se ha explicado anteriormente, pueden tomar valores enteros entre -255 y 255. Además, teniendo en cuenta que el signo no afecta al resultado, existe un número acotado de 256x256 posibles valores.

Para simplificar este paso, se han precalculado todos los posibles valores y se han almacenado en una matriz de 256x256 valores. De esta forma, en cada fila se encontrarán almacenados todos los valores de magnitud para un valor de *Gy* fijo y cualquier valor de *Gx*. De la misma forma, en cada columna se encontrarán los valores de magnitud para un valor fijo de *Gx* y cualquier valor de *Gy*. Cabe destacar que, con el objetivo de simplificar las operaciones posteriores y reducir el número de *bits*, se han redondeado los valores al valor entero más próximo.

Así, la magnitud de un determinado gradiente se obtendrá seleccionando el elemento que se encuentre en la fila igual al valor absoluto de *Gy* y en la columna igual al valor absoluto de *Gx*.

5.4.3 Cálculo orientación del gradiente

El cálculo de la orientación del gradiente se realiza aplicando la arcotangente del gradiente en dirección vertical entre el gradiente en dirección horizontal. El resultado obtenido es un ángulo que se usa posteriormente para interpolar la magnitud en el cálculo de los histogramas.

Sin embargo, el cálculo de la arcotangente supone una gran cantidad de recursos en una implementación *hardware* por lo que se ha simplificado el cálculo de la orientación del gradiente. Para ello, se va a calcular únicamente en que intervalo de los que componen los histogramas se encuentra un determinado gradiente.

Como se ha comentado anteriormente, cada histograma está conformado por 9 intervalos y, además, no se tiene en cuenta el signo del gradiente, por lo que este puede tomar valores desde 0 a 180 grados. Así, cada intervalo tiene una amplitud de 20 grados, de forma que los intervalos posibles se pueden observar en la Figura 5.11.

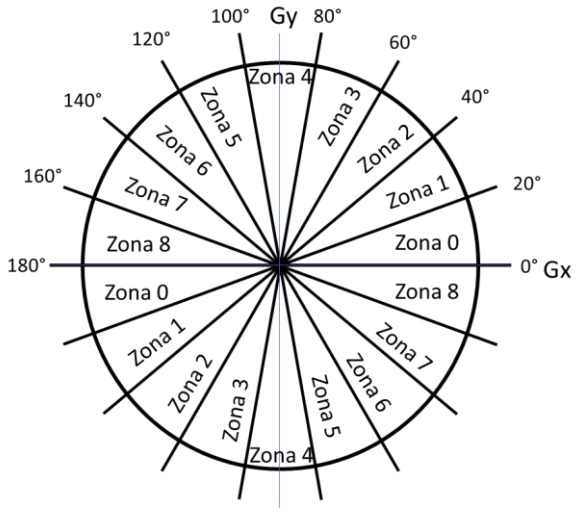


Figura 5.11: Zonas asociadas a los intervalos del histograma

La fórmula para la obtención de la orientación se puede ver de la siguiente forma:

$$Orientación = \tan^{-1}\frac{-Gy}{Gx} \rightarrow Gy = -\tan(orientación) * Gx$$

Puesto que los valores de orientación son fijos, uno por cada intervalo, es posible precalcularlos para evitar así que estos tengan que ser calculados en tiempo de ejecución. Además, teniendo en cuenta que los valores de tangente son simétricos respecto al primer cuadrante, únicamente es necesario precalcular las tangentes de los 4 intervalos del primer cuadrante: *tan(20°)*, *tan(40°)*, *tan(60°)* y *tan(80°)*.

Sin embargo, con el objetivo de reducir el número de multiplicaciones en tiempo de ejecución, es posible precalcular el valor de la tangente por cada uno de los posibles valores de *Gx*. Obviando en este momento el signo, *Gx* puede tomar valores desde 0 a 255, por lo que, para cada intervalo del primer cuadrante es necesario precalcular y almacenar 256 valores. Además, con el objetivo de simplificar las comparaciones posteriores, se van a redondear los valores al entero más cercano.

A partir del signo de *Gx* y *Gy*, se puede conocer el cuadrante de la circunferencia en la que se encuentra el gradiente. Cabe destacar que, debido a la forma de calcular el gradiente en dirección vertical es necesario cambiar el signo de dicho gradiente. Así, a modo de ejemplo,

las zonas de la cero a la tres se corresponderán con aquellos gradientes *Gx* y *Gy* que tengan signos contrarios.

Así, el procedimiento seguido para determinar a que intervalo corresponde un determinado gradiente se resume a continuación:

- Comprobación del signo de *Gx* y *Gy*: se comprueba si los gradientes tienen un mismo signo o no. En el caso de que ambos sean positivos o negativos, el gradiente podrá ser asignado a los intervalos 4, 5, 6, 7 u 8. En el caso de que ambos tengan signos contrarios, el gradiente será asignado al intervalo 0, 1, 2, 3 o 4.
- Búsqueda en los 4 vectores precalculados del valor de la tangente por *Gx*: a partir del valor absoluto de *Gx*, se busca en cada uno de los vectores el valor asociado correspondiente a la operación *tan()*Gx*.
- Comparación de los valores precalculados con *Gy*: cada uno de los valores obtenidos en el punto anterior se compara con el valor absoluto de *Gy*. El primer valor precalculado que sea mayor que el absoluto de *Gy* determinará cual es el intervalo asociado al gradiente.

En la tabla siguiente se resume el procedimiento descrito anteriormente:

Signo	Comparación	Intervalo
Signos distintos Gx positivo y Gy negativo o Gx negativo y Gy positivo	|Gy|<tan(20°)*|Gx|	0
	|Gy|<tan(40°)*|Gx|	1
	|Gy|<tan(60°)*|Gx|	2
	|Gy|<tan(80°)*|Gx|	3
	|Gy|>tan(80°)*|Gx|	4
Signos iguales Gx positivo y Gy positivo o Gx negativo y Gy negativo	|Gy|<tan(80°)*|Gx|	8
	|Gy|<tan(60°)*|Gx|	7
	|Gy|<tan(40°)*|Gx|	6
	|Gy|<tan(20°)*|Gx|	5
	|Gy|>tan(20°)*|Gx|	4

5.4.4 Generación del histograma

Como se ha explicado en el apartado anterior, la simplificación del cálculo de la orientación del gradiente únicamente permite conocer a que intervalo se ha asignado un determinado píxel. De esta forma, no es posible interpolar las magnitudes de los gradientes en función de la orientación, ya que no se conoce como de centrado en su intervalo se encuentra un determinado gradiente.

Con el objetivo de reducir el número de multiplicaciones, se ha eliminado la aplicación del filtro paso bajo gaussiano aplicado a las magnitudes de los gradientes en el algoritmo de Matlab. Este filtro tenía como objetivo disminuir la influencia de los píxeles más cercanos a los bordes de los bloques.

De esta forma, la generación de los histogramas se simplifica notablemente, limitándose a la suma de todas las magnitudes del gradiente que pertenezcan a un determinado intervalo dentro de una celda.

5.4.5 Normalización

La normalización *L2-Hys* aplicada por el algoritmo en Matlab consiste, realmente, en dos normalizaciones *L2-norm* y una umbralización. La norma L2 se calcula como la raíz

cuadrada de la suma de los cuadrados de todos los elementos a normalizar. Así, este procedimiento supone una gran cantidad de potencias y dos raíces cuadradas por cada bloque normalizado.

Para simplificar este paso, se va a normalizar los histogramas de cada bloque utilizando la norma L1. Esta norma se calcula como la suma de los valores absolutos de todos los elementos a normalizar. De esta forma, la norma se obtendrá sumando el valor de todos los intervalos que conforman los cuatro histogramas que componen un bloque.

5.4.6 Implementación en Matlab y resultados obtenidos

Con el objetivo de conocer la influencia de las simplificaciones efectuadas, se va a comparar el algoritmo simplificado con la función *extractHOGFeatures* disponible en la *toolbox* correspondiente e implementada por Matlab.

El algoritmo HOG con la función *extractHOGFeatures* recibe como parámetro una imagen de 64x128 píxeles que será procesada como una ventana de detección. La mayoría de parámetros del algoritmo HOG que han sido utilizados son los que están definidos por defecto en esta función. Sin embargo, es necesario modificar el *stride* de bloque ya que en la aplicación no se desea que se produzca solapamiento de bloques. Así, la función será llamada tal y como se muestra a continuación:

```
[valores_hog_matlab,visualization]=extractHOGFeatures(ventana_imagen,'BlockOverlap',[0 0]);
```

La función devuelve el descriptor HOG en el parámetro *valores_hog_matlab* como un vector de 1152 elementos. Además, el objeto *visualization* permite mostrar la representación del descriptor, tal y como se ha explicado anteriormente en este documento.

Para la implementación del algoritmo simplificado, se ha partido del código de la función *extractHOGFeatures* y se han aplicado sobre él las diversas simplificaciones explicadas en los apartados anteriores. Esta función recibe como parámetro una imagen de 64x128 píxeles y devuelve un descriptor en forma de vector de 1152 valores calculado con la configuración HOG indicada en el apartado 5.4.1.

Una vez calculado el descriptor, este es representado sobre la ventana procesada, de forma que es posible observar si los resultados obtenidos reflejan las direcciones predominantes de cada celda. Para ello, se ha generado un *script* en Matlab que permite obtener una representación gráfica semejante a la generada por Matlab por la función *extractHOGFeatures*. En la Figura 5.12 se pueden observar gráficamente los resultados obtenidos para dos ventanas diferentes.

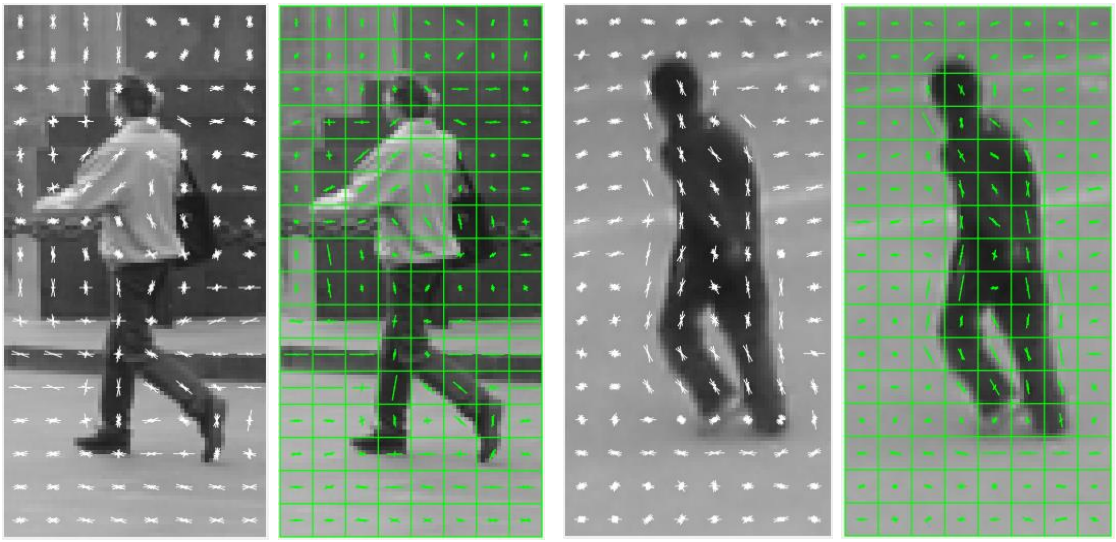


Figura 5.12: Representación del descriptor HOG generada por la función extractHOGFeatures (en blanco) y por el algoritmo simplificado (en verde)

En la Figura 5.12 se muestra en blanco la representación del descriptor calculado por la función *extractHOGFeatures* y en verde la obtenida para la función simplificada. Se puede observar que, comparando ambos, las direcciones predominantes son semejantes, señalándose los contornos visibles en las imágenes.

Para obtener una comparación cuantitativa, se han estudiado los descriptores obtenidos por ambos algoritmos. Para ello, se van a comparar cada uno de los histogramas de un descriptor con su homólogo del otro. Cada histograma va a ser comparado teniendo en cuenta únicamente el orden, de mayor a menor valor, de todos los intervalos que lo componen. Así, una vez ordenados los dos histogramas se comprueba cuantos elementos coinciden.

Los resultados obtenidos muestran que el porcentaje de acierto es elevado, en torno al 90%, cuando se comprueba únicamente la dirección predominante. Sin embargo, este porcentaje cae fuertemente cuando se comprueba el orden del resto de intervalos.

Estos resultados pueden ser debidos a que el algoritmo original premia en gran medida a los intervalos vecinos de los intervalos a los que realmente pertenece un determinado gradiente. Sin embargo, el algoritmo simplificado no distribuye la magnitud de un gradiente entre distintos intervalos, por lo que únicamente el intervalo asociado a un gradiente acumula su magnitud.

Aunque la comparación con el HOG original muestra que los descriptores obtenidos por el algoritmo simplificado difieren significativamente, los histogramas obtenidos permiten conocer las direcciones predominantes en cada celda y, por tanto, en la ventana de detección. De esta forma estos vectores pueden seguir siendo usados para detectar los bordes de una determinada imagen y, por tanto, ser empleados para detectar la presencia de personas en esta.

5.5 Implementación en SW del algoritmo HOG simplificado

Una vez explicadas las simplificaciones introducidas, se ha adaptado el algoritmo simplificado implementado en Matlab a lenguaje C para su ejecución en el procesador ARM del dispositivo Zynq.

5.5.1 Visión global en *pseudocódigo*

A continuación se muestra en *pseudocódigo* la implementación llevada a cabo en lenguaje C:

```
void hog(int inicio_ventana, float *descriptor_hog)
{
      //Declaración de constantes y variables
      for(y=0; y<128; y++)    //Se recorren todas las líneas de la
ventana de procesamiento
      {
            for(x=0; x<64; x++) //Se recorren todos los píxeles por
cada línea de la ventana
            {
                  //Cálculo del gradiente en dirección horizontal y
vertical
                  //Cálculo de la magnitud del gradiente
                  //Cálculo de la orientación del gradiente
            }
      }

      for(j=0; j<4; j++)      //Para cada bloque en dirección
horizontal de la ventana de detección
      {
            for(k=0; k<8; k++)//Para cada bloque en dirección vertical
de la ventana de detección
            {
                  //Selección de los valores de magnitud y orientación
del bloque
                  for(r=0; r<16; r++) //Para cada fila de un bloque
                  {
                        for(s=0; s<16; s++) //Para cada columna de un
bloque
                        {
                              //Generación de los 4 histogramas del
bloque
                        }
                  }
                  //Cálculo norma L1: suma de todos los elementos de
los 4 histogramas que componen el bloque procesado
                  //Normalización de cada intervalo de los histogramas
de un bloque
            }
      }
}
```

El código, tal y como puede observarse en el *pseudocódigo* anterior, puede dividirse en dos partes claramente diferenciadas: el cálculo de los gradientes y la generación y normalización de los histogramas.

Por un lado, se realiza el cálculo del gradiente para todos los píxeles que componen una ventana de detección. Para ello, primero se calculan las componentes en dirección horizontal y vertical del gradiente en cada píxel, restando el valor de luminancia de sus píxeles vecinos. A partir de estos valores, se obtiene el valor de la magnitud del gradiente resultante de la combinación de los dos anteriores, buscando en una matriz precalculada su valor asociado. Este valor es almacenado en una matriz de 64x128 elementos, en la posición asociada al píxel procesado.

De la misma forma, se calcula el intervalo asociado a la orientación del gradiente de cada uno de los píxeles de la ventana de procesamiento siguiendo el procedimiento descrito en 5.4.3. Al igual que en el caso de la magnitud, los intervalos son almacenados en una matriz de 64x128 elementos.

Así, una vez se han procesado todos los píxeles de una ventana, se dispone de dos matrices de 64x128 elementos con los valores de magnitud y el intervalo asociado al gradiente para cada uno de los píxeles.

A partir de las matrices anteriores, se generan los histogramas del gradiente de cada una de las 4 celdas que conforman cada uno de los 4x8 bloques de una ventana. Para ello, se agrupan los elementos en grupos de 8x8 y se suman los valores de magnitud de los elementos de la matriz que tengan asociados un mismo intervalo.

Una vez obtenidos los 4 histogramas de un determinado bloque, estos son normalizados por la suma de los valores de todos sus elementos. Una vez normalizados, estos histogramas son almacenados en un vector que conformará el descriptor HOG.

5.5.2 Resultados obtenidos

El algoritmo anterior se ha ejecutado en el procesador ARM del dispositivo Zynq, obteniéndose un tiempo de procesamiento por cada ventana analizada de aproximadamente 35 milisegundos.

Para comprobar el correcto funcionamiento, los descriptores obtenidos se han comparado, siguiendo los pasos explicados detalladamente en 5.6.3, con los calculados por el algoritmo simplificado en Matlab, obteniéndose los mismos resultados.

Con el objetivo de poder comparar estos tiempos de ejecución, en los siguientes apartados se explican detalladamente todos los aspectos necesarios para la implementación del algoritmo HOG simplificado en *hardware* programable.

5.6 Implementación en HW programable

En este apartado se va a implementar el algoritmo HOG simplificado en una función C++ a partir de la cual se generará un bloque HW. Este bloque *hardware* podrá ser posteriormente añadido en una aplicación diseñada con la herramienta Vivado.

Para ello, se utilizará la herramienta Vivado HLS desarrollada por Xilinx y que permite describir en lenguaje de alto nivel una determinada función y posteriormente sintetizarla para ser implementada posteriormente en un dispositivo HW programable.

5.6.1 Codificación algoritmo HOG simplificado en HLS

En los siguientes apartados se explicará detalladamente el código C++ utilizado para implementar el algoritmo HOG simplificado, tal y como se ha explicado en apartados anteriores.

5.6.1.1 Visión global en pseudocódigo

Con el objetivo de entender globalmente el código utilizado, en este apartado se va a comentar de forma general las diferentes partes que lo conforman. A continuación se muestra el *pseudocódigo* de la función desarrollada:

```
void hog(int inicio_ventana, volatile unsigned char *imagen, volatile
ap_ufixed<16, 1> *descriptor_hog)
{
      //Declaración de constantes y variables
      for(j=0; j<32; j++)//Para cada bloque en la ventana de detección
{
            for(i=0; i<18; i++)//Lectura de las 18 filas de un bloque
de píxeles
            {
                  #pragma HLS PIPELINE
                  //Lectura de una línea de 18 píxeles
            }
            for(y=0; y<16; y++)//Se recorren todas las líneas del
bloque leído
            {
                  #pragma HLS UNROLL
                  for(x=0; x<16; x++)//Se recorren todos los píxeles
por cada línea
                  {
                        #pragma HLS PIPELINE II=2
                        //Cálculo del gradiente en dirección horizontal
y vertical
                        //Cálculo de la magnitud del gradiente
                        //Cálculo de la orientación del gradiente
                        //Generación del histograma para cada celda del
bloque
                  }
            }
            for(m=0; m<36; m++)//Se recorre cada intervalo de los 4
histogramas que conforman un bloque: 9 intervalos por histograma por 4
histogramas por bloque
            {
                  #pragma HLS PIPELINE II=1
                  //Normalización de cada intervalo de los histogramas
de un bloque
                  //Escritura en memoria de cada intervalo normalizado
            }
      }
}
```

Observando el *pseudocódigo* anterior se observa que este puede ser dividido en 3 partes, repetidas todas ellas 32 veces, una por cada bloque que compone una ventana de detección. La primera de las partes se encarga de la lectura de cada uno de dichos bloques para su posterior procesado. Esta lectura se realiza por líneas, necesitando 18 lecturas por bloque, tal y como se explicará posteriormente. Con el objetivo de indicar a la herramienta que segmente al máximo posible esta lectura, se emplea el *pragma PIPELINE*.

La segunda de las partes en la que se puede dividir este algoritmo se centra en el cálculo de los gradientes de cada uno de los píxeles que componen un bloque y su posterior clasificación en histogramas. Para ello, se emplean dos bucles que permiten recorrer los

16x16 píxeles que deben ser procesados. Cada iteración del bucle más externo, que recorre las líneas de los bloques, es ejecutada en paralelo lo que permite acelerar su ejecución a costa de un mayor empleo de recursos. Esto es indicado a la herramienta HLS mediante el *pragma UNROLL*. El bucle más interno, que recorre cada uno de los píxeles de cada fila, es segmentado con una cadencia de 2 ciclos. Esta cadencia viene impuesta por las limitaciones del *hardware* en los accesos de lectura y escritura en el momento del cálculo del histograma.

Por último, la tercera parte se encarga de la normalización de los 4 histogramas de un bloque y su escritura en memoria DDR. Este bucle es segmentado con una cadencia de 1 ciclo de reloj.

5.6.1.2 Definición de la función top

La función *top* en HLS representa la función de mayor jerarquía del diseño. Por ello, esta función debe recoger todas las interfaces que tendrá el bloque IP y que serán empleadas para el intercambio de datos con el resto del diseño en Vivado.

Para llevar a cabo el procesamiento de la imagen, es necesario leer esta desde las direcciones de memoria DDR donde se almacena. Para ello, la función *top* en HLS va a recibir como parámetros dos elementos: un entero y un puntero volátil de tipo *unsigned char*.

Además, una vez llevado a cabo el procesamiento, el descriptor HOG debe ser escrito en otra dirección de memoria DDR. Para ello se utilizará otra interfaz definida como un puntero volátil de tipo coma fija sin signo.

Teniendo en cuenta todo lo anterior, la definición de la función será la siguiente:

```
void hog(int inicio_ventana, volatile unsigned char *imagen, volatile
ap_ufixed<16, 1> *descriptor_hog)
```

5.6.1.2.1 Interfaces AXI4

Los accesos a memoria DDR, tanto de lectura como de escritura, se realizarán utilizando el protocolo AXI4. Este protocolo está diseñado para realizar transmisiones de grandes cantidades de datos a memoria.

Para poder utilizar este protocolo, es necesario definir como parámetro de la función *top* un puntero del tipo de datos que serán transmitidos. Este puntero representa la dirección base de memoria desde la cual se realizará la transferencia de datos. En esta aplicación, se van a utilizar dos interfaces AXI4, una utilizada para leer de memoria los píxeles de la imagen y otra para escribir en memoria el descriptor HOG obtenido.

En el caso de la interfaz de lectura, como los datos leídos pueden tomar valores de 0 a 255, el tipo de datos configurado es *unsigned char*. En el caso de la interfaz de escritura, se ha estipulado un tipo de datos de coma fija sin signo. En este caso, cada elemento ocupará 2 *bytes*, dejando para la parte entera un único *bit*.

La etiqueta *volatile* es necesaria ya que, sin ella, Vivado HLS interpreta que el valor apuntado no cambiará a lo largo de la ejecución de la función. En este caso, puesto que se van a leer múltiples píxeles en cada ejecución de la función *top*, el valor apuntado cambiará múltiples veces siendo, por tanto, necesario añadir este atributo. De la misma forma, serán escritos múltiples valores en memoria en cada ejecución de la función por lo que también es necesario añadir este atributo al puntero que representa la interfaz de escritura.

Para poder utilizar el protocolo AXI4, es necesario indicar al programa HLS que los punteros recibidos como parámetros de entrada deben ser implementados como interfaces AXI4 de tipo maestro. Para ello se emplea la directiva *interface* tal y como se muestra a continuación:

```
#pragma HLS INTERFACE m_axi port=imagen offset=slave depth= 1310720
bundle=INPUT_IMAGE
#pragma HLS INTERFACE m_axi port=descriptor_hog offset=slave
depth=1152 bundle=OUT_HOG
```

El primer elemento indica el tipo de interfaz que se desea implementar. En este caso, se va a utilizar una interfaz AXI4 de tipo máster a la cual le corresponde la etiqueta *m_axi*. La siguiente opción, *port*, indica el nombre de la variable recibida como parámetro que debe asociarse a la interfaz AXI. El parámetro *offset* permite configurar la dirección base de memoria desde donde se leerá la imagen y se escribirá el descriptor. En el caso de una interfaz AXI4 esta opción puede tomar 3 valores distintos:

- *Off*: es la opción por defecto y supone que la dirección de acceso es la 0x00000000.
- *Direct*: esta opción crea un puerto en el bloque IP en el cual se puede fijar, desde el proyecto en Vivado, un *offset* a partir del cual se accederá a memoria.
- *Slave*: genera una interfaz *AXI4-Lite* que permite configurar el *offset* desde *software*.

En este caso, se ha elegido esta última opción puesto que permite una mayor libertad en el diseño. El parámetro *depth* especifica el máximo número de muestras leídas/escritas cada vez que se llama a la función en el *testbench* por lo que únicamente es relevante para el proceso de cosimulación C/RTL. Por último, la opción *bundle* se emplea para agrupar diferentes señales en una misma interfaz que toma el nombre asignado.

5.6.1.2.2 Interfaz AXI-Lite

Además de los punteros explicados en el apartado anterior, la función *top* recibe como argumento un entero denominado *inicio_ventana*. Esta variable se utilizará para seleccionar en una imagen la esquina superior izquierda de la ventana a procesar, tal y como posteriormente se explicará.

Para escribir esta variable se va a utilizar una interfaz *AXI4-Lite* que es una versión simplificada de la interfaz AXI4. Este protocolo solo permite transacciones de un único dato, a diferencia del protocolo completo que permite el envío de ráfagas de datos.

Para indicar al programa HLS que esta variable será accedida utilizando el protocolo *AXI-Lite* es necesario utilizar la directiva *interface* con el parámetro *s_axilite*, tal y como se indica a continuación:

```
#pragma HLS INTERFACE s_axilite port=inicio_ventana bundle=AXI_Lite
```

Como se ha comentado en el apartado anterior, las interfaces AXI4 se han configurado con la opción *slave*, lo que permite configurar la dirección de memoria desde *software* a través de una interfaz *AXI-Lite*. Con el objetivo de utilizar la misma interfaz que la utilizada por la variable *inicio_ventana* se han utilizado las siguientes expresiones:

```
#pragma HLS INTERFACE s_axilite port=imagen bundle=AXI_Lite
#pragma HLS INTERFACE s_axilite port=descriptor_hog bundle=AXI_Lite
```

Por último, la implementación de un bloque utilizando la herramienta HLS lleva asociada una serie de *drivers* que permiten el control de dicho bloque desde *software*. A modo de ejemplo, estas funciones permiten al programador indicar el inicio de operación del bloque

o consultar si este ha finalizado o no su ejecución. Para poder utilizarlas, es necesario incluir la siguiente sentencia:

```
#pragma HLS INTERFACE s_axilite port=return bundle=AXI_Lite
```

5.6.1.3 Lectura de memoria DDR del bloque a procesar

Para poder llevar a cabo el procesado de la ventana de detección, es necesario leer de memoria fragmentos de esta y almacenarlos en BRAM. En este caso, las lecturas se realizan de bloques completos ya que con estos datos se puede generar cada uno de los histogramas normalizados que componen el descriptor HOG.

5.6.1.3.1 Organización en la lectura de bloques de píxeles

Cabe destacar que, aunque los bloques tienen un tamaño 16x16 píxeles, es necesario almacenar matrices de 18x18 ya que, como se ha explicado anteriormente, para el cálculo del gradiente se emplean los píxeles vecinos. Así, los píxeles de los bordes pueden conocer el valor que toman sus vecinos para poder calcular su gradiente en cada punto del bloque de 16x16 píxeles. En la Figura 5.13 se muestra gráficamente como son leídos y almacenados los píxeles que componen cada uno de los bloques de una ventana de detección.

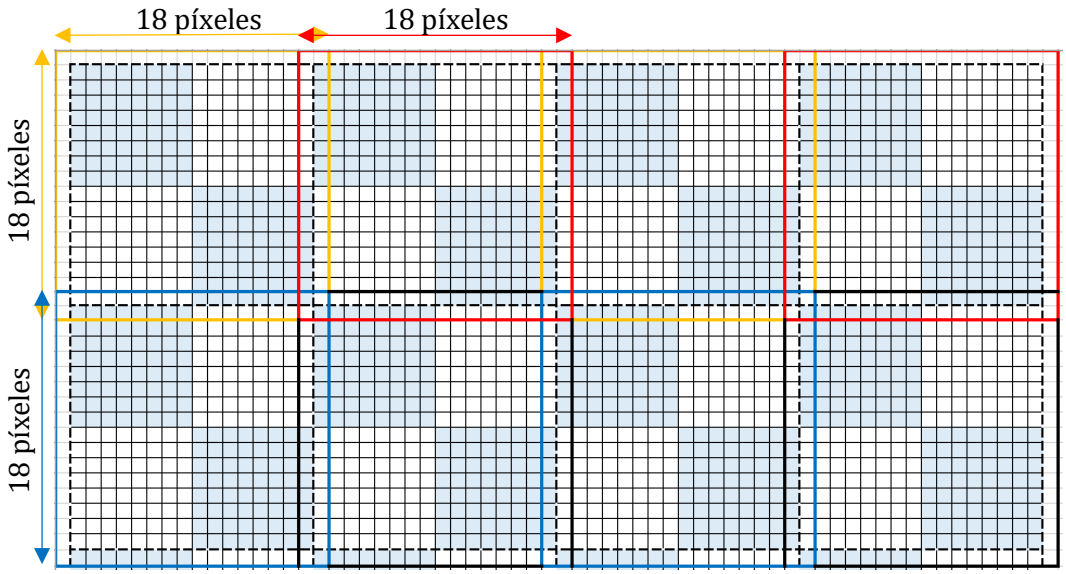


Figura 5.13: Píxeles leídos por bloque

De esta forma, cada bloque que compone la ventana de detección es leído y almacenado en una matriz de 18x18 valores de tipo *unsigned char*. Este tipo de datos es suficiente ya que cada uno de los valores de luminancia de un determinado píxel están almacenados en un único *byte* y, por tanto, puede tomar valores de 0 a 255.

Con el objetivo de aumentar la paralelización y disminuir los cuellos de botella que se pueden producir en la lectura y escritura de bloques de memoria BRAM, se puede indicar al sintetizador HLS que divida un determinado *array* en otros más pequeños o, incluso, que cada elemento sea implementado como un registro. De esta forma, se eliminan los posibles cuellos de botella a costa de incrementar los recursos necesarios. Para ello, se emplea la directiva *array_partition* como se muestra a continuación:

```
unsigned char buffer_imagen[18][18];
#pragma HLS ARRAY_PARTITION variable=buffer_imagen complete dim=1
```

El argumento *variable* indica el *array* que se desea particionar, en este caso, *image_buffer0*.

El siguiente argumento "complete" indica el tipo de partición que se desea aplicar. Este apartado puede tomar una de las siguientes opciones:

- *Cyclic*: Crea *arrays* de menor tamaño rellenándolos de forma cíclica con los elementos del *array* original. De esta forma, elementos consecutivos del *array* a particionar se almacenan en los diferentes *arrays* nuevos.
- *Block*: Genera *arrays* menores rellenándolos con consecutivos elementos del *array* original.
- *Complete*: se descompone el *array* en elementos individuales. Para el caso de un *array* con una única dimensión, esta opción convierte cada elemento en un registro individual.

El campo *dim* especifica en que dimensión de un *array* multidimensional debe realizarse el particionado. Este valor puede tomar cualquier valor entero de 0 a N, siendo N el número de dimensiones del *array* a particionar. Cabe destacar que, si se indica el valor 0, todas las dimensiones serán particionadas.

Para los tipos de particionado *cyclic* y *block* es necesario especificar en cuantos *arrays* se va a particionar la variable. Para ello, se emplea el campo *factor* seguido del número de *arrays* que se deben generar. En la Figura 5.14 se puede observar un ejemplo para un *factor* igual a 2.

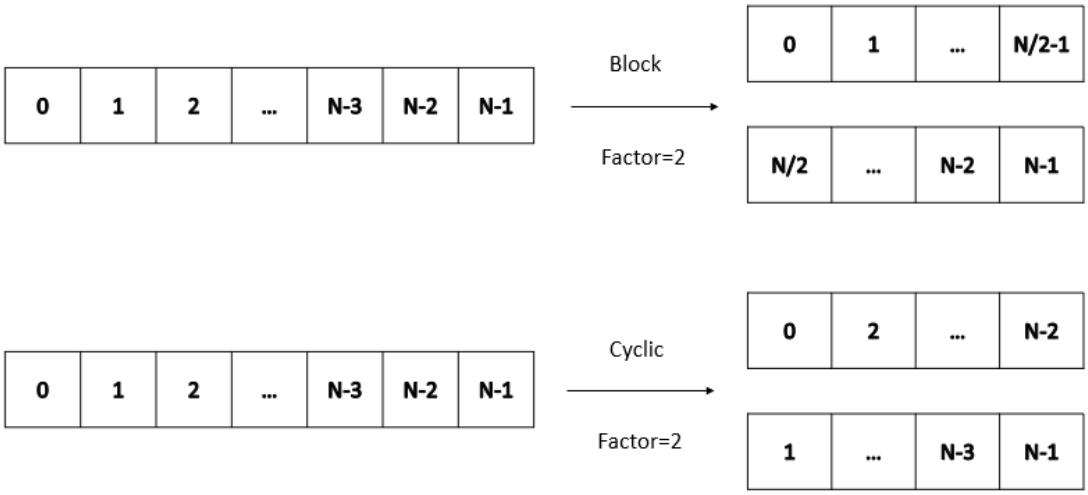


Figura 5.14: Ejemplo array_partition de tipo block y cyclic

En este caso concreto, especificando *complete dim=1*, la matriz *buffer_imagen[18][18]* se divide en 18 vectores de 18 elementos cada uno, tal y como puede observarse en la Figura 5.15. De esta forma, cada uno de los vectores puede ser accedido de forma concurrente, aumentando la paralelización en la ejecución del algoritmo.

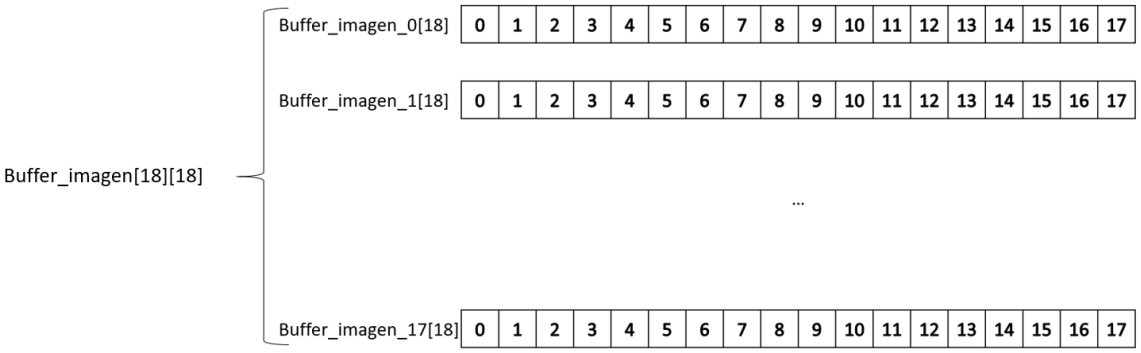


Figura 5.15: Ejemplo array_partition complete

5.6.1.3.2 Lectura de datos a través de bus AXI4

Existen dos métodos para llevar a cabo la lectura o escritura de datos a través de un bus AXI4: transferencia individual o modo ráfaga. En el caso de la transferencia individual, HLS lee o escribe un único dato para una determinada dirección. Por el contrario, el modo *burst* o ráfaga permite realizar múltiples transferencias de datos de forma secuencial, es decir, a partir de una determinada dirección se leen o escriben N datos. Este último método permite obtener un mayor rendimiento en la transferencia de datos al tener que comunicar sólo la dirección de inicio, no una dirección por cada dato.

Aunque para el caso de la aplicación que se está desarrollando no sería necesario, se va a diseñar este bloque considerando que la imagen almacenada en memoria DDR es una de las obtenidas por la aplicación desarrollada en el capítulo 2. En esa aplicación las imágenes son almacenadas por filas en memoria DDR en formato YUV 4:2:2.

La lectura de cada píxel que compone un bloque debe llevarse a cabo realizando 18 transferencias de datos, cada una de ellas leyendo una fila de 18 píxeles. Sin embargo, debido al formato YUV 4:2:2 en el cual está almacenada la imagen, los valores de luminancia de cada píxel se ubican en las direcciones pares de memoria, estando los valores de crominancia en las direcciones impares. De esta forma, aunque únicamente hagan falta los valores de luminancia "Y", el modo *burst* fuerza la lectura de los 36 valores que definen los 18 píxeles de cada bloque. En este caso, el número de *bytes* que deben ser leídos en cada ráfaga es 36, correspondiente a 18 píxeles formados por dos canales (luminancia y crominancia) de 1 *byte* cada uno. Así, una vez leídos todos, se desecharán los correspondientes a los valores de crominancia "UV" almacenando únicamente los valores de luminancia para su posterior uso.

El fragmento de código utilizado para almacenar un bloque para su posterior procesamiento se muestra a continuación:

```
for(int i=0;i<18;i++)
{
#pragma HLS PIPELINE
      memcpy(buffer_imagen_temp,(const unsigned char*)imagen +
inicio_ventana + i*2560 + (j%8)*40960 + (j/8)*32,36*sizeof(unsigned
char));
      for(int k=0;k<18;k++)
      {
            buffer_imagen[i][k]=buffer_imagen_temp[k*2];
      }
}
```

La función *memcpy* implementa una lectura en modo ráfaga de cada una de las líneas que componen un bloque. El primer parámetro que recibe la función es el *array* donde se almacenarán los datos leídos de memoria DDR a través del bus AXI4. El segundo parámetro indica la dirección a partir de la cual debe llevarse a cabo la lectura y el último, el número de *bytes* a leer.

En este caso, la dirección base de lectura viene dada por el argumento de entrada *imagen* que apunta a la primera dirección de memoria DDR donde se encuentra almacenada la imagen. A esta dirección base hay que sumarle el *offset*, recibido por el bus *AXI-Lite*, que identifica la esquina superior izquierda de la ventana de detección.

El resto de los sumandos permiten controlar que bloque debe ser leído y que línea de dicho bloque:

- *i*2560*: *offset* que permite seleccionar cuál de las 18 filas de cada bloque debe ser leída. La constante 2560 corresponde al número de píxeles por columna de la imagen (1280) por el número de *bytes* que compone cada píxel (2, uno de luminancia y otro de crominancia).
- *(j%8)*40960*: *offset* que determina cuál es la línea de la imagen donde comienza el bloque a procesar. La variable *j* es una variable que toma valores entre 0 y 31 y se encarga de indicar cual es el bloque que debe ser procesado. La constante 40960 corresponde al número de *bytes* por fila de la imagen (1280x2) por el número de filas de un bloque.
- *(j/8)*32*: *offset* que determina cuál es la columna de la imagen donde comienza el bloque a procesar. La constante 32 corresponde al número de *bytes* por fila en un bloque (16*2).

Una vez se ha realizado la lectura, las muestras de crominancia almacenadas en los elementos impares de *buffer_imagen_temp* son desechadas, conservándose únicamente los valores de luminancia en la matriz *buffer_imagen* que será posteriormente procesada.

5.6.1.4 Cálculo del gradiente

El cálculo del gradiente se puede dividir en 3 apartados claramente diferenciados: cálculo de las componentes horizontal y vertical del gradiente, obtención de la magnitud del gradiente y obtención de su orientación.

5.6.1.4.1 Cálculo de las componentes horizontal y vertical del gradiente

Tal y como se ha explicado en la parte teórica del algoritmo HOG, el cálculo del gradiente en una imagen mide como varía la luminosidad de píxeles adyacentes en dirección horizontal y vertical. De esta forma, para cada píxel se calculan dos valores de gradiente, uno en sentido horizontal y otro en sentido vertical. En ambos casos, el valor de luminosidad del píxel estudiado no es utilizado, afectando en el valor del gradiente únicamente los 4 píxeles vecinos.

El gradiente en dirección horizontal, *Gx*, se calcula como la resta del valor de luminancia del píxel a la derecha del elemento estudiado menos la luminancia del píxel a la izquierda. De la misma forma, el gradiente en dirección vertical, *Gy*, se calcula como la resta del píxel situado en la fila inferior menos el píxel situado en la fila superior del píxel actual. En la Figura 5.16 se puede apreciar este procedimiento visualmente:

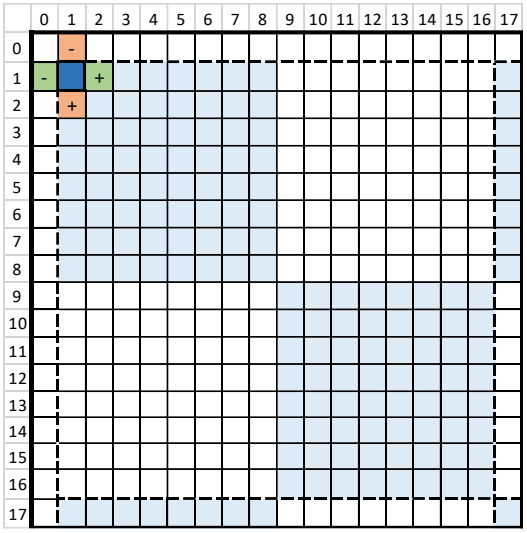


Figura 5.16: Cálculo de los histogramas vertical y horizontal

Codificado en lenguaje C/C++, el cálculo de las componentes *Gx* y *Gy* puede expresarse como se muestra a continuación:

```
Gx = buffer_imagen[y+1][x+2] - buffer_imagen [y+1][x]; //Cálculo
componente horizontal
Gy = buffer_imagen [y+2][x+1] - buffer_imagen [y][x+1];    //Cálculo
componente vertical
```

A partir de las componentes del gradiente horizontal y vertical, se puede calcular la magnitud y la orientación del gradiente resultante.

5.6.1.4.2 Obtención de la magnitud del gradiente

Como se ha explicado anteriormente, el cálculo de la magnitud del gradiente se va a sustituir por una búsqueda de dicho valor en una matriz precalculada. La matriz recoge todos los posibles valores de la operación módulo, redondeados al valor entero más próximo. Puesto que el valor absoluto de las componentes *Gx* y *Gy* pueden tomar valores de 0 a 255, la matriz tendrá unas dimensiones de 256x256 elementos.

Con el objetivo de disminuir el número de recursos, se va a utilizar un tipo de dato entero, *ap_int*, que permite al programador indicar el número de *bits* usados para almacenar un determinado dato. Además, como todos los datos son positivos, es posible seleccionar el tipo de dato como *unsigned*, *ap_uint*, evitando así utilizar el *bit* más significativo para almacenar el signo. En este caso, puesto que el mayor valor de magnitud es 361, el número de *bits* necesarios es 9.

Puesto que la matriz nunca va a ser modificada en tiempo de ejecución, es posible declarar esta de tipo constante con la expresión *const*. Este tipo de datos son almacenados en memorias de tipo ROM y son inicializados en el momento de la implementación, lo que permite ahorrar ciclos de reloj en su inicialización.

```
const ap_uint<9> mod[256][256]={{0, 1, … }, {}, …};
```

La obtención del valor de magnitud asociado a los gradientes *Gx* y *Gy* se reduce a una búsqueda en la matriz *mod*. Puesto que los gradientes pueden tomar valores negativos, el primer paso es calcular el valor absoluto de ambos valores. Una vez obtenidos, estos valores son los utilizados para seleccionar la fila y la columna donde se encuentra el valor de magnitud buscado.

```
absx = abs(Gx);   //Obtención del valor absoluto de Gx
absy = abs(Gy);   //Obtención del valor absoluto de Gy
mag = mod[absy][absx];  //Búsqueda del valor de magnitud asociado a los
valores de Gx y Gy
```

5.6.1.4.3 Obtención de la orientación del gradiente

Para la obtención de la orientación del gradiente, según se ha explicado en el apartado 5.4.3, es necesario almacenar 4 vectores de 256 elementos cada uno con los valores precalculados de la tangente por todos los posibles valores del valor absoluto de *Gx*.

Al igual que en el apartado anterior, se ha utilizado el mínimo número de *bits* necesarios para almacenar cada elemento, dependiendo en cada *array* del valor máximo recogido. De la misma forma, se ha declarado cada *array* de tipo *const* ya que sus elementos únicamente van a ser leídos, no siendo modificados en ningún momento durante la ejecución del algoritmo.

```
const ap_uint<7> tan20[256]={0,0,…,93};//Array de 256 valores
precalculados: tan(20)*abs(Gx)
const ap_uint<8> tan40[256]={0,1,…,214};//Array de 256 valores
precalculados: tan(40)*abs(Gx)
const ap_uint<9> tan60[256]={0,2,…,442};//Array de 256 valores
precalculados: tan(60)*abs(Gx)
const ap_uint<11> tan80[256]={0,6,…,1446};//Array de 256 valores
precalculados: tan(80)*abs(Gx)
```

Para conocer en que intervalo se encuentra un determinado gradiente, el primer paso es determinar en que cuadrante de la circunferencia se encuentra. Para conocerlo, se utiliza el signo de las dos componentes que lo definen, *Gx* y *Gy*. Como se ha justificado en el apartado 5.4.3, si el signo de las componentes es diferente, el gradiente podrá pertenecer a los intervalos 0, 1, 2, 3 o 4. Por el contrario, si el signo es el mismo, ambos positivos o ambos negativos, el gradiente podrá pertenecer a los intervalos 4, 5, 6, 7 u 8.

Para conocer el signo de las componentes *Gx* y *Gy* se ha utilizado la clase *sign* de la biblioteca *ap_int* que devuelve un 1 si el número es negativo y un 0 si es positivo.

Para determinar en que intervalo se encuentra la orientación de un gradiente se compara el valor absoluto de *Gy* con el resultado de la tangente del intervalo por el valor absoluto de *Gx*. Esta última operación es la que se encuentra precalculada en los vectores anteriormente descritos, por lo que únicamente es necesario realizar una búsqueda en estos.

El código utilizado en HLS para la obtención simplificada del intervalo en el que cae el valor del gradiente se muestra a continuación:

```
if((Gx.sign()==1 && Gy.sign()==0) || (Gx.sign()==0 &&
Gy.sign()==1))//Comprobación de los signos de las componentes del
gradiente
{     //Si los signos de Gx y Gy son distintos
      if (absy < tan20[absx])       {intervalo = 0;}
      else if (absy < tan40[absx]) {intervalo = 1;}
      else if (absy < tan60[absx]) {intervalo = 2;}
      else if (absy < tan80[absx]) {intervalo = 3;}
      else {intervalo = 4;}
}
else  //Si los signos de GX y Gy son iguales
{
      if (absy < tan20[absx]) {intervalo = 8;}
```

```
      else if (absy < tan40[absx]) {intervalo = 7;}
      else if (absy < tan60[absx]) {intervalo = 6;}
      else if (absy < tan80[absx]) {intervalo = 5;}
      else {intervalo = 4;}
}
```

5.6.1.5 Generación del histograma

La generación del histograma se reduce a la acumulación de la magnitud de un determinado gradiente al intervalo correspondiente, según su orientación, del histograma asociado a la celda en la que se encuentre el píxel en estudio.

Los 4 histogramas que pertenecen a un determinado bloque son almacenados en el vector *histogramas*. Este vector está formado por 36 elementos, 9 por cada uno de los 4 histogramas que conforman un bloque.

En la declaración del *array*, este se ha definido de tipo estático anteponiendo la expresión *static*. Una variable estática es inicializada en el momento de implementación del bloque y, además, mantiene su valor entre las diferentes ejecuciones de la función. En este caso, todos los intervalos deben ser inicializados a 0 antes de realizar la acumulación por lo que en este caso se ahorran algunos ciclos de reloj.

El pragma *resource* permite al programador elegir qué tipo de memoria se va a utilizar para almacenar los datos. En este caso, se ha seleccionado el tipo LUTRAM de 2 puertos con el objetivo de reducir la latencia de acceso.

```
static ap_uint<16> histogramas[36];
#pragma HLS RESOURCE variable=histogramas core=RAM_2P_LUTRAM
```

Dependiendo de la celda en la que se ubique un píxel dentro del bloque que se está procesando, su gradiente influirá en uno de los 4 histogramas que componen un bloque. Para conocer en que celda se ubica el píxel procesado se comprueba el valor que tienen las variables *x* e *y*. Estas variables, que toman valores de 0 a 15, son las utilizadas para recorrer cada columna (*x*) y cada fila (*y*) del bloque leído de memoria.

La forma más simple de lograrlo es consultando el valor del cuarto *bit* menos significativo de cada una de ellas. En el caso de la variable *y*, si dicho *bit* es un 0, el píxel leído pertenece a las primeras 8 filas y si, por el contrario, es un 1, pertenecerá a alguna de las filas siguientes. Del mismo modo, en el caso de la variable *x*, si el *bit* es un 0, el píxel pertenecerá a las primeras 8 columnas y, si es un 1, a alguna de las siguientes columnas. Para consultar el valor de un determinado *bit* de una variable de tipo *ap_uint*, se puede utilizar la clase *test*, indicando la posición del *bit* que se desea conocer. Esta función devuelve un 1 cuando el *bit* sea un 1 y un 0 en caso contrario.

Tal y como se explicó anteriormente y se muestra en la Figura 5.17, el orden de los histogramas que componen un bloque en el descriptor HOG sigue primero la dirección vertical y, posteriormente, la horizontal. Así, el primer histograma corresponderá a la celda superior izquierda del bloque, el segundo a la inferior izquierda, el tercero a la superior derecha y, el último, a la inferior derecha.

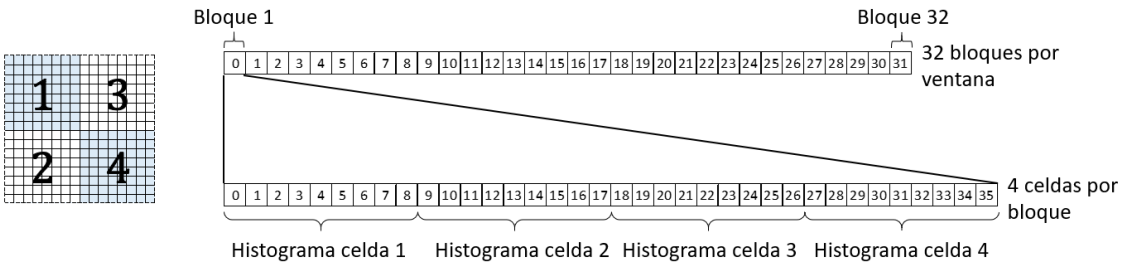


Figura 5.17: Orden de los histogramas en el descriptor HOG

De esta forma, para actualizar el histograma con el valor de magnitud del gradiente son necesarios 3 parámetros:

- *x.test(3)*18*: Si el píxel pertenece a las celdas 3 o 4, introduce un *offset* de 9*2 intervalos.
- *y.test(3)*9*: Si el píxel pertenece a las celdas inferiores, 2 o 4, introduce un *offset* de 9 intervalos.
- *Intervalo*: Indica a que intervalo, del 0 al 8, pertenece la orientación del gradiente.

```
histogramas[x.test(3) * 18 + y.test(3) * 9 + intervalo] += mag;
//Generación de los histogramas
sum += mag; //Acumulación de todas las magnitudes de los gradientes que
pertenecen a un mismo bloque
```

Por último, la variable *sum* almacena la suma de todas las magnitudes que pertenecen a un mismo bloque. Este valor, denominado también norma L1, será utilizado posteriormente en el proceso de normalización de los histogramas.

5.6.1.6 Normalización y escritura de los histogramas en memoria DDR

Una vez se han obtenido los 4 histogramas de un bloque, es necesario normalizarlos y escribirlos en memoria utilizando la interfaz AXI4.

La normalización de cada histograma se lleva a cabo mediante la división de cada intervalo entre la norma L1. Con el objetivo de simplificar esta operación de división, se van a utilizar datos en coma fija, implementados en HLS mediante la biblioteca *ap_fixed*. Puesto que la división supone una gran cantidad de ciclos en una implementación *hardware* y que el valor divisor es constante para todos los intervalos, primero se ha calculado la inversa y, posteriormente, se multiplica esta por cada intervalo.

Una vez realizada la normalización, el intervalo es escrito en memoria a través del puerto AXI4. Además de con el empleo de la función *memcpy*, se pueden realizar escrituras en modo *burst* empleando un bucle *for* con el *pragma pipeline*. En este caso, a la dirección base, referenciada por el propio puntero *descriptor*, es necesario sumarle dos *offset*. El primer *offset*, *j*36*, indica cual es el bloque cuyos histogramas están siendo escritos. El segundo *offset*, *m*, incrementa en una unidad la dirección de escritura en memoria.

Por último, cabe destacar que, una vez escrito un determinado intervalo en memoria DDR, el valor del intervalo debe ser puesto a 0 para que la siguiente iteración del algoritmo comience con los histogramas inicializados a 0.

```
inversa=(ap_ufixed<16,1>)1/sum;
for (int m = 0; m < 36; m++) //Se recorren todos los intervalos que
componen los 4 histogramas
```

```
{
#pragma HLS PIPELINE II=1
      descriptor_hog[j*36+m]=histogramas[m]*inversa;
      histogramas[m]=0; //Inicialización de los intervalos a 0
}
```

5.6.2 Comprobación de funcionamiento mediante *testbench*

Para comprobar el correcto funcionamiento de la función diseñada, se ha creado un banco de pruebas o *testbench* cuyo *pseudocódigo* se muestra a continuación.

```
Mat im_grises = imread(argv[1], CV_LOAD_IMAGE_GRAYSCALE); //Lectura y
conversión a escala de grises de una imagen de tamaño 1280x1024
píxeles indicada como argumento
im_yuv422=conversión_formato_yuv422(im_grises); //Conversión a formato
YUV422

unsigned int numWindowsX = ((1280 -2 -64) / 8) + 1; //Cálculo del
número de ventanas en dirección horizontal
unsigned int numWindowsY = ((1024 -2 -128) / 16) + 1; //Cálculo del
número de ventanas en dirección vertical

for (y = 0; y < numWindowsY; y++)
{
      for (x = 0; x < numWindowsX; x++)
      {
            Inicio_ventana = y*2*16*1280+x*8*2; //Se calcula el píxel
de inicio de la ventana a procesar
            hog(inicio_ventana,im_yuv422, descriptor_hog); //Llamada a
la función top

            //Escritura de los descriptores calculados en un archivo de
texto para su posterior comparación
            fprintf(f, "Ventana %d = [", ventana);
            for(int val = 0; val < 1152; val++)
            {
                  fprintf(f, "%f ", (float)descriptor_hog[val]);
            }
            fprintf(f, "]\n");
            ventana++;
      }
}
```

En relación a los datos de entrada, la función HOG espera poder acceder a una imagen que se encuentre almacenada en un *array*. Así, el primer paso consiste en la lectura de una imagen con la función *imread* de la biblioteca OpenCV. Como se ha explicado anteriormente, para el procesado HOG únicamente es necesario tener la imagen en escala de grises por lo que, tras leer la imagen, esta se almacena en una única matriz del tamaño de la imagen, en este caso de 1024x1280.

Sin embargo, debido al formato YUV 4:2:2 esperado por la función HOG diseñada y a la forma de *array* esperada, es necesario transformar la matriz de la imagen leída en un *array* unidimensional de 2x1024x1280, tal y como se encontrará la imagen en memoria DDR una vez el diseño HLS sea exportado y añadido al proyecto completo. En este vector, los elementos pares deben almacenar todos los valores de luminancia de la imagen leída y

cualquier valor en los elementos impares puesto que estos serán desechados en el procesamiento HOG.

Cada ejecución de la función *top* se encarga de procesar una ventana de detección de 64x128 píxeles. Con el objetivo de obtener un gran número de descriptores que puedan ser posteriormente comparados con los calculados por el algoritmo simplificado implementado en Matlab, será necesario llamar a la función HOG un gran número de veces, fijando el inicio de la ventana de procesamiento en diferentes puntos de la imagen completa. Esta posición se indica a la función *top* a través del primer argumento que esta recibe, *inicio_ventana*.

Para cada una de las ventanas de detección, la función HOG devuelve un vector con el descriptor HOG calculado. Este descriptor es leído desde el *testbench* y escrito en un archivo de texto para su posterior comparación.

Desarrollado el *testbench*, se puede llevar a cabo una simulación en C del funcionamiento de la función HOG para comprobar si se obtienen los resultados esperados.

Los valores obtenidos en la simulación C se han comparado con los calculados mediante el algoritmo simplificado implementado en Matlab, obteniéndose un error máximo en el rango de 10^{-4}. Este error es el introducido por el empleo de la coma fija en el algoritmo implementado en HLS, por lo que se considera que los resultados son correctos.

5.6.3 Síntesis, cosimulación y exportación del bloque IP

Una vez que se ha comprobado el correcto funcionamiento de la función, se debe llevar a cabo el proceso de síntesis. En la Figura 5.18, se muestran los resultados obtenidos tras la síntesis de la función HOG.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	6.67	6.12	0.83

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
33345	33345	33346	33346	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	4190
FIFO	-	-	-	-
Instance	0	2	1502	1936
Memory	40	-	304	81
Multiplexer	-	-	-	2093
Register	-	-	3127	33
Total	40	2	4933	8333
Available	280	220	106400	53200
Utilization (%)	14	~0	4	15

Figura 5.18: Estimación de rendimiento y recursos tras síntesis HLS

En la estimación de los recursos utilizados se puede observar que el uso de memoria BRAM es relativamente elevada. Esto es debido principalmente a la necesidad de almacenar una

gran cantidad de valores precalculados evitando así tener que calcularlos durante la ejecución, logrando una mayor velocidad de procesamiento.

Posteriormente, con el bloque ya sintetizado, se procede a realizar la cosimulación C/RTL. Esta simulación hace uso de la implementación RTL y del *testbench* por lo que, al igual que en la simulación C, se genera un archivo de texto con los descriptores HOG. Sin embargo, a diferencia de la simulación C, estos descriptores han sido generados teniendo en cuenta la implementación RTL por lo que los resultados obtenidos serán los esperables una vez que este bloque sea implementado en una tarjeta de desarrollo. Cabe destacar que los valores obtenidos en la cosimulación son idénticos a los obtenidos en la simulación C.

Una vez que se ha comprobado que los resultados son los esperados, se procede a exportar el bloque IP para que posteriormente pueda ser incorporado a un repositorio de Vivado.

5.7 Comprobación de funcionamiento del bloque IP desarrollado

En este apartado se va a diseñar una aplicación que procesará una imagen con el algoritmo HOG y devolverá los descriptores calculados. Para ello, la imagen a procesar será recibida por un puerto Ethernet, a través del cual, también se enviarán los descriptores HOG calculados. En la Figura 5.19, se muestra un diagrama donde se resume la aplicación que se va a emplear para comprobar el correcto funcionamiento del bloque IP.

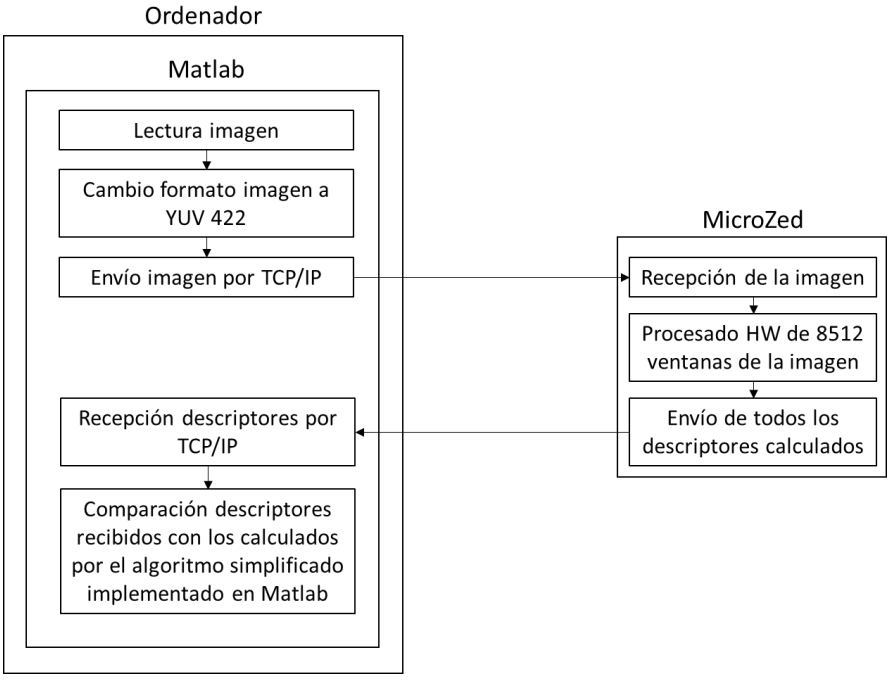


Figura 5.19: Esquema de la aplicación desarrollada

Las imágenes recibidas, al igual que las obtenidas en el capítulo 2, tendrán un tamaño de 1024x1280 píxeles y tendrán un formato YUV 4:2:2. Cada imagen será procesada múltiples veces, desplazando la ventana de detección por esta y, por tanto, generando múltiples descriptores HOG, uno por cada ventana analizada.

Además, para comprobar el correcto funcionamiento del algoritmo implementado, se compararán todos los descriptores calculados con los obtenidos con el algoritmo simplificado implementado en Matlab.

5.7.1 Diseño hardware en Vivado

Una vez se ha creado un proyecto nuevo, el primer paso es añadir el bloque IP diseñado al repositorio de Vivado. Una vez hecho esto, el bloque HLS puede ser encontrado junto al resto de módulos IP y añadido al diseño.

Tal y como se ha comentado en el apartado anterior, el bloque HLS dispone de dos puertos maestros AXI4 y un puerto esclavo *AXI-Lite*.

El puerto *AXI-Lite* debe conectarse a uno de los puertos maestros de propósito general del dispositivo Zynq. A su vez, los dos puertos AXI4 deben ser conectados a un puerto esclavo de alto rendimiento. Ambas conexiones se efectúan a través de bloques *AXI Interconnect*.

Además de las interfaces anteriores, el bloque Zynq debe ser configurado para generar una señal de reloj de 150 MHz que será utilizada por el bloque HLS como señal de sincronismo.

Por último, los periféricos *UART1* y *ENET0* deben estar habilitados. El controlador *UART1* es el utilizado para gobernar el puerto serie y en esta aplicación se utilizará para informar del estado de ejecución de la aplicación. Por su parte, el controlador *ENET0* es el encargado de controlar las comunicaciones Ethernet de la tarjeta MicroZed.

El diseño *hardware* utilizado en Vivado para esta aplicación se puede observar en la Figura 5.20.

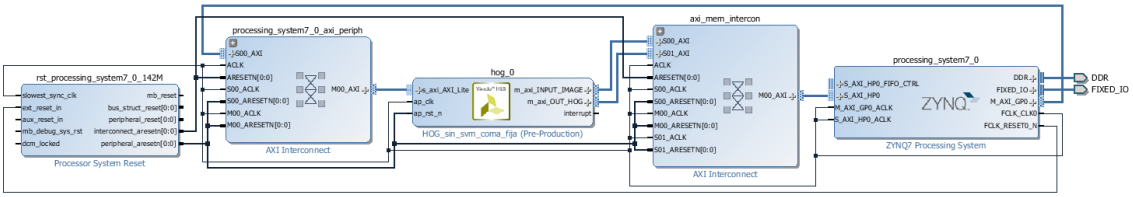


Figura 5.20: Diseño hardware algoritmo HOG

5.7.2 Diseño software en Vivado SDK

Una vez se ha generado el archivo *bitstream* que recoge la programación del *hardware*, es necesario llevar a cabo la programación del código que será ejecutado por el procesador ARM que integra el dispositivo Zynq.

Para implementar la comunicación Ethernet se va a hacer uso de la biblioteca LwIP o *Lightweight IP*. LwIP es una implementación en código libre de la pila de protocolos TCP/IP pensada principalmente para sistemas empotrados debido a su bajo uso de recursos y su sencilla configuración.

5.7.2.1 Inicializaciones

El empleo de la pila de protocolos LwIP se lleva a cabo utilizando una API denominada *Raw API* que reúne todo lo necesario para el empleo de esta pila sin necesidad de un sistema operativo. La puesta en funcionamiento de esta pila de protocolos requiere de una serie de inicializaciones que configuran todos los parámetros de esta, como la dirección IP utilizada, el puerto de escucha y escritura o el empleo de DHCP.

El bloque HLS también lleva asociado un conjunto de *drivers* que reúnen todas las funciones que permiten controlar el funcionamiento del bloque. En el caso de esta aplicación, estas funciones se encuentran declaradas en el archivo *xhog.h* que debe ser incluido en el código para poder utilizarlas posteriormente.

Antes de poder utilizar el bloque IP, este debe ser inicializado. Esta inicialización se lleva a cabo con la siguiente instrucción:

```
XHog_Initialize(&HOG0,XPAR_HOG_0_DEVICE_ID);
```

El primer argumento de la llamada a esta función es una estructura de datos que se usará posteriormente para el control de este bloque y donde se recoge la dirección de memoria donde se encuentran mapeados los registros del bus *AXI-Lite*. El segundo argumento es el nombre dado al bloque IP en el archivo *xparameters.h* donde se recoge todo el mapeado en memoria del sistema y que se emplea para conocer dicha dirección.

5.7.2.2 Lectura de imagen, procesamiento y escritura de descriptores HOG

Una vez el sistema esta inicializado correctamente, este espera recibir una imagen por el puerto Ethernet. Para ello, se comprueba constantemente si se están recibiendo datos y, cuando se reciben, estos son almacenados en memoria.

Una vez que se ha recibido la imagen completa, es el momento de iniciar su procesamiento. El primer paso es configurar los buses AXI4 del bloque IP con las direcciones de memoria desde la cual se debe leer la imagen y en la que se escribirán los descriptores calculados. Para ello, se hace uso de dos de las funciones incluidas en los *drivers* generados por HLS:

```
XHog_Set_imagen(&HOG0,(u32)0x18000000);//Configuración de la dirección
base de memoria DDR donde se almacena la imagen a leer
XHog_Set_descriptor_hog(&HOG0,(u32)dir_hog);//Configuración de la
dirección de memoria donde se escribirán los descriptores calculados
por el algoritmo HOG
```

La función *XHog_Set_imagen* se emplea para configurar la dirección de memoria desde donde el bus AXI4 leerá la imagen. Esta función está asociada a la configuración del bus AXI4 con un *offset* de tipo *slave* en el código HLS, tal y como se explicó en el apartado 5.6.1.2.1, que permite configurar esta dirección desde *software* empleando el bus *AXI-Lite*. En este caso, se indica la dirección 0x18000000 de memoria ya que esa dirección es donde la imagen es almacenada tras ser recibida vía Ethernet.

De la misma forma, la función *XHog_Set_descriptor_hog* permite configurar la dirección a partir de la cual se escribirán los descriptores calculados.

Como se detallará en el siguiente capítulo, la búsqueda de personas en una imagen supone recorrerla desplazando las ventanas de detección y decidiendo para cada una de ellas si se localiza una persona o no. Con el objetivo de realizar un procesamiento similar, cada imagen de 1024x1280 píxeles debe ser procesada múltiples veces, desplazando la ventana de detección de 128x64 píxeles por ella. El desplazamiento de la ventana debe realizarse superponiendo las ventanas entre sí, con el objetivo de cubrir todas las áreas donde podría encontrarse una persona.

En este caso se ha estipulado un *stride* de ventana de 8 píxeles en dirección horizontal y 16 píxeles en dirección vertical. Así, el número de ventanas en cada dirección se obtendrá teniendo en cuenta el número de píxeles de la imagen, el tamaño de la ventana y el solapamiento de estas con las siguientes operaciones:

```
numWindowsX = ((1280 - 2 - 64) / stridex) + 1;//Número de ventanas en
dirección horizontal
```

```
numWindowsY = ((1024 - 2 - 128) / stridey) + 1;//Número de ventanas en
dirección vertical
```

Una vez determinado el número de ventanas, es necesario ejecutar el bloque *hardware* una vez para cada una de ellas. Como se ha explicado anteriormente, el bloque debe recibir también el *offset* dentro de la imagen que indica la esquina superior izquierda de la ventana de detección a procesar. Para ello, se han utilizado dos bucles anidados, uno recorriendo la dirección vertical y otro la horizontal. Utilizando estos bucles, el tamaño de la imagen y el *stride* de ventana aplicado, es posible determinar el píxel donde comienza la ventana a procesar. Cabe destacar que la multiplicación por un factor 2 es debido al formato YUV 4:2:2 en el que se encuentra la imagen almacenada en memoria.

```
inicio = y*2*stridey*1280+x*2*stridex; //Cálculo del offset de la
ventana a procesar dentro de la imagen completa.
XHog_Set_inicio_ventana(&HOG0, inicio); //Configuración del offset en
el bloque HLS
```

Una vez se han configurado todos los parámetros que necesitaba el bloque HLS, es necesario indicarle que puede empezar su ejecución. Para ello se emplea la función *XHog_Start*, que se incluye en los *drivers* del bloque, tal y como puede observarse en la línea siguiente:

```
XHog_Start(&HOG0); //Inicio procesamiento HOG
```

Una vez el procesamiento ha comenzado, se espera a que este termine su ejecución antes de iniciar el procesamiento de una nueva ventana. Para conocer cuando la ejecución ha terminado, se emplea la función *XHog_IsDone* que devuelve un nivel alto cuando esto sucede.

```
while(!XHog_IsDone(&HOG0)) //Se comprueba si el procesamiento ha
terminado
{
}
```

Cuando el procesamiento de la función ha terminado, el descriptor calculado para la ventana procesada se encuentra almacenado en la dirección de memoria fijada por *dir_hog*. Con el objetivo de que la siguiente iteración no sobrescriba los resultados de la anterior, es necesario modificar la dirección base de escritura, incrementando esta en 1152 valores.

```
dir_hog = dir_hog + 1152;
XHog_Set_descriptor_hog_V(&HOG0,(u32)dir_hog);
```

Una vez se han procesado todas las ventanas de una imagen, se envían todos los descriptores almacenados en DDR a un ordenador empleando el puerto Ethernet, de forma similar al procedimiento de lectura de la imagen.

5.7.3 Verificación de funcionamiento

Una vez terminado el diseño de la aplicación, se procede a cargarla en placa para su ejecución en un entorno real.

5.7.3.1 Conexionado

Para el funcionamiento correcto de esta aplicación, se deben realizar una serie de conexiones entre la tarjeta MicroZed y un ordenador con el cual se realizarán las transferencias vía Ethernet. Un esquema con el conexionado necesario para esta aplicación se muestra en la Figura 5.21.

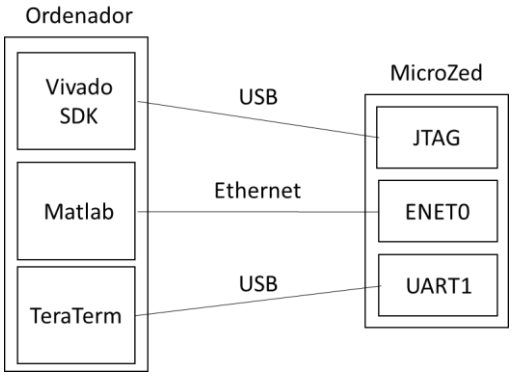


Figura 5.21: Esquema conexionado aplicación cálculo HOG

5.7.3.2 Programación y ejecución

Una vez el diseño *software* compila correctamente, el ejecutable es generado por la herramienta SDK. Este ejecutable junto con el archivo *bitstream* de configuración *hardware* deben ser cargados a la tarjeta MicroZed. Para ello, al igual que en las aplicaciones anteriores, se hará uso de un dispositivo JTAG. Una vez realizada la programación, el sistema espera la recepción de una imagen vía la interfaz Ethernet.

5.7.3.2.1 Aplicación en Matlab

La comunicación Ethernet desde el punto de vista del ordenador será controlada por un *script* desarrollado en Matlab. Este *script* hace uso de la función *tcpip* de la *toolbox Instrument Control* que permite el envío y recepción de datos empleando estos protocolos. Esta función recibe una dirección IP y un puerto a los cuales realiza las escrituras y lecturas de datos.

La función de Matlab se encarga de leer una imagen y convertirla al formato YUV 4:2:2. Posteriormente esta imagen es enviada a la tarjeta MicroZed para que esta la procese. Una vez enviada, el *script* espera a que termine dicho procesamiento para leer todos los descriptores calculados.

Una vez recibidos todos los descriptores, estos son comparados con los calculados anteriormente por el algoritmo simplificado en Matlab para la misma imagen que ha sido enviada a la tarjeta MicroZed.

5.7.3.2.2 Resultados obtenidos

La ejecución de la aplicación anterior devuelve como resultado el error cometido entre los valores recibidos y los calculados por el algoritmo simplificado en Matlab. El máximo error cometido para los 8512 vectores de 1152 elementos es del orden de 10^{-4}. Este error se asocia al empleo de la coma fija en el algoritmo implementado en HLS frente al empleo de operaciones en coma flotante en Matlab.

En relación a la velocidad de procesamiento, se logran unos tiempos de ejecución de aproximadamente 0,7 ms por ventana de 64x128 píxeles procesada. Este tiempo es sustancialmente más reducido al obtenido con la ejecución del algoritmo en el procesador ARM del dispositivo Zynq.

En la siguiente tabla se recogen resumidos todos los resultados obtenidos a lo largo de este capítulo en las diferentes implementaciones estudiadas.

Tiempo de ejecución	Algoritmo HOG original en SW	Algoritmo HOG simplificado en SW	Algoritmo HOG simplificado en HW
Tiempo por ventana de 64x128 píxeles	1 segundo	35 ms	700 µs

Se puede observar que el factor de aceleración del algoritmo implementado en *hardware* frente al ejecutado en el procesador ARM es de 50.

Capítulo 6: Detección de personas mediante HOG y SVM

En este capítulo se pretende utilizar el bloque de cálculo del descriptor HOG, previamente generado en el capítulo 5, en un sistema enfocado en la detección de personas en imágenes. El objetivo de este detector no es tanto obtener un buen resultado de detección de personas desde el punto de vista algorítmico, sino analizar cómo es la implementación en HW/SW del algoritmo de detección SVM, las alternativas de implementación y la modificación del sistema en función de las especificaciones deseadas. Estas especificaciones pueden ser, por ejemplo, tener resultados de detección a una alta tasa de refresco, o generar detecciones con una alta fiabilidad.

El descriptor HOG obtenido tras analizar una ventana de procesamiento permite conocer las direcciones predominantes en cada región del espacio, detectando así los bordes de los objetos que se encuentran en la imagen. Así, aprovechando la información de todos los bordes detectados se puede identificar si un determinado objeto se encuentra en la ventana analizada o no.

La identificación de un objeto en una imagen a partir de un descriptor de características es denominado problema de clasificación. Un problema de clasificación consiste en elegir a cuál de las categorías posibles se asocia una determinada observación. En el caso de la detección de objetos en imágenes, las categorías posibles son los propios objetos a detectar y las observaciones los diferentes descriptores obtenidos de una imagen.

Estos problemas pueden ser solucionados empleando algoritmos de aprendizaje supervisado o no supervisado. Estos algoritmos de aprendizaje generan una función de decisión a partir de datos de entrenamiento que están previamente clasificados en el caso del aprendizaje supervisado o que no lo están en el caso del aprendizaje no supervisado.

En el caso de este trabajo, se pretende detectar la existencia de personas en las imágenes capturadas por el sensor PYTHON, empleando para ello el algoritmo HOG implementado en el capítulo anterior. Así, la aplicación del algoritmo HOG sobre diferentes ventanas de la imagen completa serán las observaciones a partir de las cuales se determinará la existencia o no de una persona en cada una de las ventanas. En el caso de que se determine su existencia, se marcará su posición sobre la imagen, de forma que sea fácilmente distinguible su localización.

6.1 Máquinas de vectores soporte – SVM

Una máquina de vector soporte o SVM (*Support Vector Machines*) es un tipo de algoritmo de aprendizaje supervisado que busca un hiperplano que separe de forma óptima dos clases a diferenciar. Este tipo de clasificador también suele ser denominado clasificadores de margen máximo ya que el hiperplano buscado será aquel que tenga la mayor distancia posible de todos los puntos que se encuentren cercanos a él.

En la Figura 6.1 se puede observar un ejemplo sencillo para dos dimensiones donde los datos que deben ser clasificados se encuentran representados en el plano X-Y. El algoritmo SVM intenta encontrar una recta que separe ambas clases de forma que la distancia, entre las muestras más cercanas a ella sea lo mayor posible. Estas muestras son denominadas *vectores soporte* y la distancia que los separa es denominada *margen*.

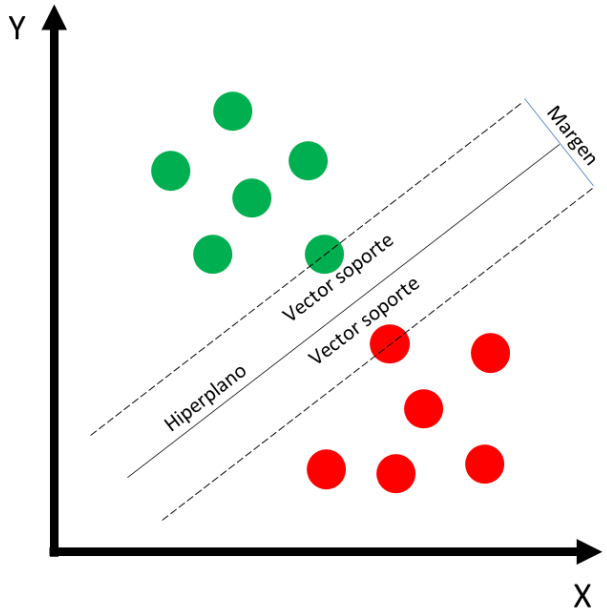


Figura 6.1: Máquina vector soporte

Aunque la base teórica en las que se basan las máquinas vector soporte fue originalmente establecida por Vapnik en 1963, no fue hasta la década de los 90 cuando las SVM tomaron la forma actual [14]. Originalmente pensadas para la resolución de problemas de clasificación con dos únicas clases y cuyos datos fueran separables linealmente, actualmente también son utilizadas para resolver problemas multiclase y no lineales. Para ello, se propuso la adición de una función *kernel* que permite realizar proyecciones a espacios de características de mayor dimensión donde el conjunto de datos sí sea separable. Esto supone que el empleo de *kernels* de tipo gaussianos o polinómicos permiten emplear las SVM como clasificadores no lineales.

Sin embargo, en el caso de este trabajo, la función *kernel* utilizada es lineal por lo que la decisión de clasificación se basa en el resultado de una combinación lineal de sus características. Para ello, el hiperplano que separa correctamente todas las muestras del conjunto de entrenamiento es definido por el vector de pesos y el término independiente que conjuntamente definen la función discriminante lineal del clasificador. Este vector de pesos tiene la misma longitud que el número de características de los descriptores utilizados en el proceso de aprendizaje.

Así, la decisión de clasificación se toma a partir del sumatorio de la multiplicación de cada elemento del vector de pesos por cada elemento correspondiente del descriptor calculado para una determinada ventana. Una vez sumado a ese valor el termino independiente o umbral se obtiene un valor que, si es positivo se clasifica como de una categoría y, si es negativo, como de la otra. Cabe destacar que cuanto mayor sea el valor absoluto del resultado, mayor es la distancia con el hiperplano y, por tanto, es mayor la confianza de la clasificación realizada.

6.2 Aplicación de la SVM para la detección de personas

Como se ha descrito anteriormente, el algoritmo SVM puede ser utilizado de forma satisfactoria para detectar personas utilizando como observaciones el descriptor HOG obtenido del procesamiento de una ventana. Para ello, es necesario realizar previamente un entrenamiento de la SVM y comprobar su desempeño mediante un *test* de evaluación.

6.2.1 Entrenamiento SVM

El entrenamiento de una SVM supone, como punto de partida, el empleo de un conjunto de imágenes que se encuentren previamente clasificadas en las dos clases que se desean diferenciar. En el caso de este trabajo, puesto que se desea conocer la existencia de peatones las dos clases posibles son *persona* o *no persona*.

Como conjunto de imágenes se ha utilizado el *dataset* de personas de INRIA [15]. Este *dataset* está formado por más de 3500 imágenes positivas, donde se encuentra una persona, de 64x128 píxeles, divididas en dos conjuntos, uno de entrenamiento y otro de *test*. Un ejemplo de una de las ventanas positivas disponibles se muestra en la Figura 6.2. Además, incluye aproximadamente 1600 imágenes negativas, sin personas en ellas, de diferentes tamaños e igualmente divididas en dos conjuntos, uno de entrenamiento y otro de prueba.

Figura 6.2: Imagen positiva dataset INRIA

Para proceder a entrenar una SVM, el primer paso es obtener los descriptores de las 2416 ventanas positivas del conjunto de entrenamiento. Para ello, se va a utilizar el algoritmo simplificado HOG, explicado en el capítulo anterior e implementado en Matlab. De la misma forma, se deben calcular los descriptores HOG de ventanas obtenidas del conjunto de imágenes de entrenamiento negativas. En este caso, se han calculado 25000 descriptores HOG de ventanas negativas.

Se va a hacer uso de la función *fitcsvm* incluida en Matlab que permite entrenar de forma muy sencilla una máquina de vector soporte. Para ello, únicamente es necesario pasarle como argumentos todos los descriptores calculados y una lista de etiquetas asociadas a estos donde se indique si el descriptor ha sido calculado a partir de una ventana positiva o negativa. En relación a los descriptores, estos deben ser almacenados por filas, de forma que cada uno de ellos será una observación y cada columna una característica. Por último, en relación al *kernel* utilizado, como se ha comentado anteriormente, se emplea el tipo lineal.

A continuación se muestra en *pseudocódigo* utilizado en Matlab para realizar el entrenamiento SVM a partir de los descriptores HOG obtenidos mediante el algoritmo simplificado:

```
pos=calculo_hog_ventanas_positivas; %Cálculo de los descriptores HOG
de las ventanas positivas del conjunto de entrenamiento
neg=calculo_hog_ventanas_negativas; %Cálculo de los descriptores HOG
de ls ventanas negativas del conjunto de entrenamiento
descriptores=[pos; neg]; %Concatenación de todos los vectores
obtenidos
labels=[ones(size(pos,1), 1); zeros(size(neg,1), 1)]; %Etiquetado por
clases de los vectores concatenados: clase 1 para personas; clase 0
para no personas
svm=fitcsvm(descriptores, labels, 'KernelFunction', 'linear');
%Entrenamiento de la SVM con un kernel lineal
```

Una vez se ha ejecutado la función *fitcsvm* se obtiene una estructura, mostrada en la Figura 6.3, que recoge, entre otros muchos parámetros, el vector de pesos, en la variable *Beta*, y el umbral, en la variable *Bias*, que se emplearán para determinar la detección o no de una persona en una ventana según el descriptor obtenido de esta.

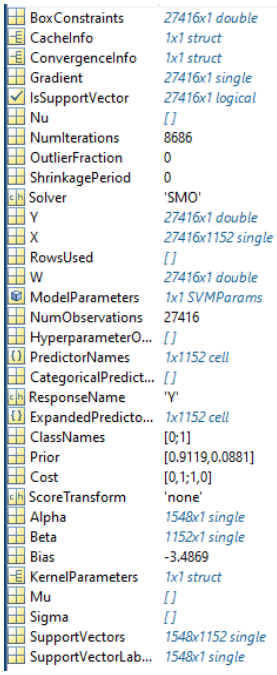


Figura 6.3: Estructura de datos obtenida tras la ejecución de la función fitcsvm

6.2.2 Test de evaluación de la SVM entrenada

Una vez realizado el entrenamiento, es necesario conocer el rendimiento del clasificador obtenido. Para ello, se van a utilizar el conjunto de imágenes de *test* disponibles en el *dataset* INRIA.

El conjunto de *test* de muestras positivas está formado por 1132 ventanas. Cada una de estas ventanas ha sido procesada por el algoritmo HOG simplificado implementado en Matlab. Posteriormente, a los descriptores obtenidos se les ha aplicado la función discriminante de la SVM entrenada y se ha comprobado en cuantas ocasiones el resultado era mayor que cero, detectando así una persona y clasificando, por tanto, la ventana correctamente. En el caso de este algoritmo, se ha conseguido clasificar correctamente 1039 de las 1132 ventanas

positivas disponibles, obteniendo un 91,78% de acierto. Este valor suele ser denominado exhaustividad o *recall* y también suele ser expresado de la forma contraria, indicando la tasa de falsos negativos frente al total de ventanas positivas estudiadas. Este parámetro es denominado *miss rate* y, en este caso, supone un 8,22%.

Por otra parte, se ha calculado el número de falsos positivos para las imágenes de *test* negativas del *dataset*. Para ello, cada una de las imágenes negativas ha sido dividida en múltiples ventanas y se han procesado todas ellas con el algoritmo HOG simplificado. Al igual que en el caso anterior, la función discriminante ha sido aplicada a cada uno de ellos y se ha comprobado en cuantas ocasiones el resultado era mayor que cero. En estos casos, se considera que la ventana ha sido incorrectamente clasificada. En este caso, se han procesado un total de 11840 ventanas negativas, de las cuales 106 han sido falsos positivos y, por tanto, clasificadas como *persona*. Esto supone un porcentaje de falsos positivos (FPPW) del 0,9%.

Para evaluar distintos modelos de clasificación, frecuentemente se emplean una serie de métricas obtenidas a partir de los resultados anteriores [16]. A continuación, se comentan 2 de las más utilizadas:

- Exactitud (*Accuracy*): fracción de predicciones que se realizaron correctamente, tanto positivas como negativas, del total de predicciones realizadas.

$$Exactitud = \frac{Número\ de\ predicciones\ correctas}{Número\ total\ de\ predicciones} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{1039 + 11734}{1039 + 11734 + 106 + 93} = 98{,}47\%$$

- Precisión: identifica cuantas predicciones positivas son correctas frente al total de clasificadas como positivas.

$$Precisión = \frac{Número\ de\ predicciones\ positivas\ correctas}{Número\ total\ de\ predicciones\ positivas} = \frac{TP}{TP + FP} = \frac{1039}{1039 + 106} = 90{,}74\%$$

A tenor de los resultados anteriores, se considera que el detector SVM entrenado con los descriptores HOG obtenidos con la ejecución del algoritmo simplificado puede ser empleado para la detección de personas en imágenes.

6.3 Implementación SW del detector SVM

En este apartado se van a aunar los desarrollos explicados en los capítulos 3 y 5 de forma que las imágenes capturadas por el sensor PYTHON puedan ser procesadas por el algoritmo HOG simplificado e implementado en HW. Además, una vez este último haya procesado una determinada ventana, se determinará en *software* la existencia o no de persona empleando para ello el descriptor HOG obtenido y el vector de pesos de la SVM entrenada anteriormente.

6.3.1 Aplicación en Vivado

Para la realización de este apartado se parte del diseño HW explicado detalladamente en el capítulo 3. En este capítulo se comentó que las imágenes capturadas por el sensor eran almacenadas temporalmente en *buffers* en memoria DDR antes de ser leídas para ser mostradas en un monitor empleando una interfaz HDMI.

El bloque IP desarrollado en el capítulo anterior será configurado para acceder a dichos *buffers* de memoria de forma que lea, empleando el protocolo *AXI*, las diferentes ventanas

que conforman cada una de las imágenes. Una vez procesada cada ventana, el bloque IP escribirá en memoria DDR el descriptor HOG calculado, permitiendo acceder desde *software* a dichos valores para determinar, empleando los pesos SVM, la detección o no de una persona en la ventana.

Como se ha comentado en el capítulo anterior, este bloque tiene dos puertos *AXI*, uno para la lectura de los píxeles que componen las ventanas de detección y otro para escribir en memoria el descriptor HOG calculado. Ambos puertos deben ser conectados al bloque *AXI Interconnect* ya existente en el diseño, teniendo únicamente que añadir 2 puertos maestros a este.

Para el control del bloque a procesar, el módulo IP tiene una interfaz *AXI-Lite* que debe ser conectada, junto al resto de interfaces del mismo tipo que tiene el diseño, al bloque *AXI Interconnect* disponible para tal función.

En relación a la señal de reloj, esta se conectará a la denominada *FCLK_CLK1* generada por el dispositivo Zynq a 150 MHz. Por último, el *reset* se conectará a la señal generada para este propósito por el bloque *Proccesor System Reset* asociado a la señal de reloj utilizada.

En la Figura 6.4 se puede observar una parte del diseño desarrollado en Vivado una vez que el bloque IP descrito en el capítulo anterior es añadido al proyecto explicado en el capítulo 3.

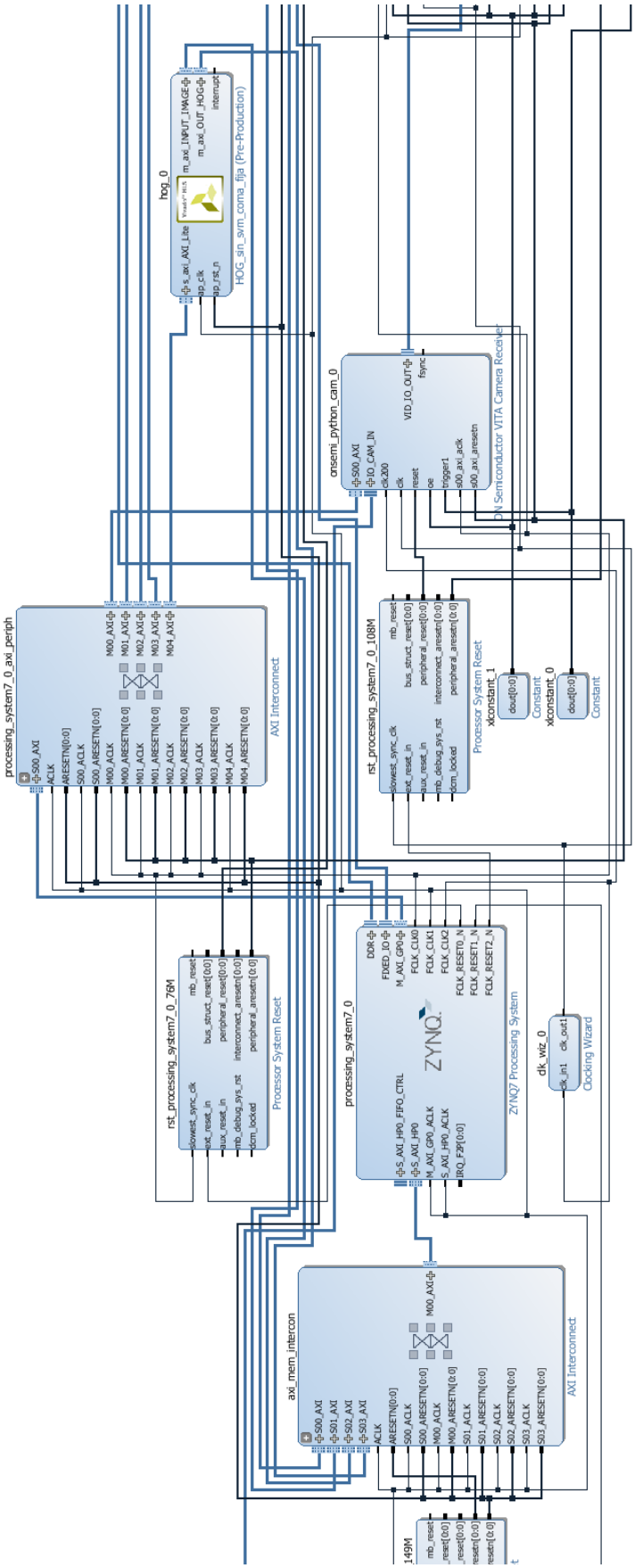


Figura 6.4: Adición del bloque HOG al diseño de captura y visualización de imágenes

Tal y como se indicó en el capítulo 3, el bloque OSD va a ser el encargado de señalar las personas que sean detectadas dibujando un rectángulo sobre ellas. Para ello, se debe habilitar una segunda capa de tipo *Internal Graphics Controller* y configurada con una memoria de al menos 100 instrucciones de tipo *box*. La configuración utilizada para esta aplicación se muestra en la Figura 6.5.

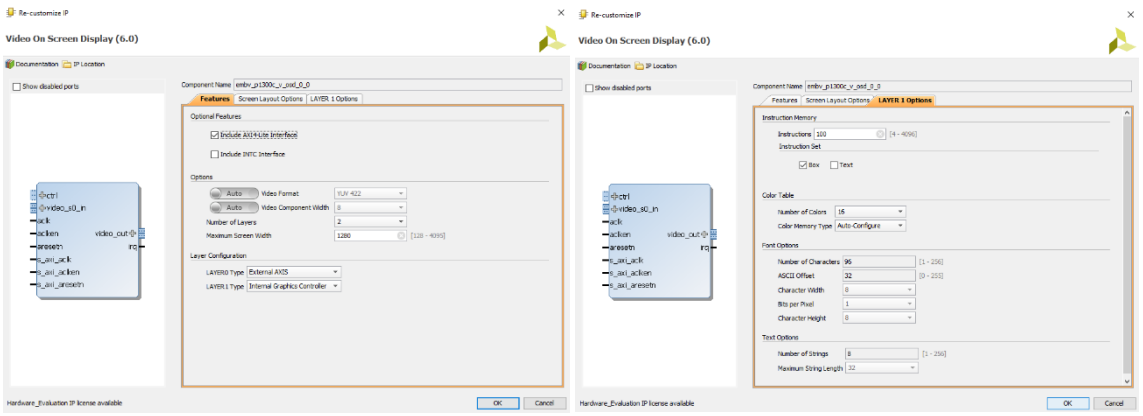


Figura 6.5: Configuración del controlador gráfico del bloque OSD

Antes de proceder a la implementación del diseño HW es necesario implementar un sistema que prevenga que el *buffer* que está siendo leído por el bloque HOG sea escrito por el módulo VDMA con una nueva imagen. Para ello, se va a configurar este bloque de forma que el canal de escritura en memoria, cuyo papel en la tarea de sincronismo es del tipo maestro dinámico tal y como se explicó anteriormente, tenga dos esclavos. De esta forma, antes de la escritura en un determinado *buffer* de memoria DDR, este canal deberá comprobar que dos *buffers* están siendo leídos por el canal de lectura y por el bloque HOG.

Como también se ha comentado anteriormente, los puertos de entrada de sincronismo de este bloque son internos cuando los dos canales son configurados como una pareja maestro-esclavo. Para que esta conexión pueda ser accesible desde el exterior, es necesario introducir la siguiente expresión en la consola TCL de Vivado.

```
set_property -dict [list CONFIG.C_S2MM_GENLOCK_NUM_MASTERS {2}]
[get_bd_cells axi_vdma_1]
```

Una vez ejecutada la línea anterior, se genera una señal *S2MM_FRAME_PTR_IN* de 12 *bits* por la cual se puede indicar al *Dynamic Master* que *buffers* están siendo leídos para evitar que este los sobrescriba. La indicación de cada *buffer* accedido se realiza mediante un código *Gray* de 6 *bits*, por lo que es necesario concatenar las dos señales antes de conectarlas a dicho puerto.

Una de las señales concatenadas será directamente la señal *MM2S_FRAME_PTR_OUT[5:0]* que genera el canal de lectura del bloque VDMA y que indica el *buffer* que está leyendo. Sin embargo, la otra señal dependerá del *buffer* que este siendo accedido por el bloque HOG. Para controlar esto, se va a hacer uso del bloque IP *AXI GPIO* que permite controlar el estado de dos señales desde *software* empleando un bus *AXI-Lite*.

Este bloque es configurado con dos buses de 6 *bits*, uno de entrada y otro de salida, tal y como puede observarse en la Figura 6.6.

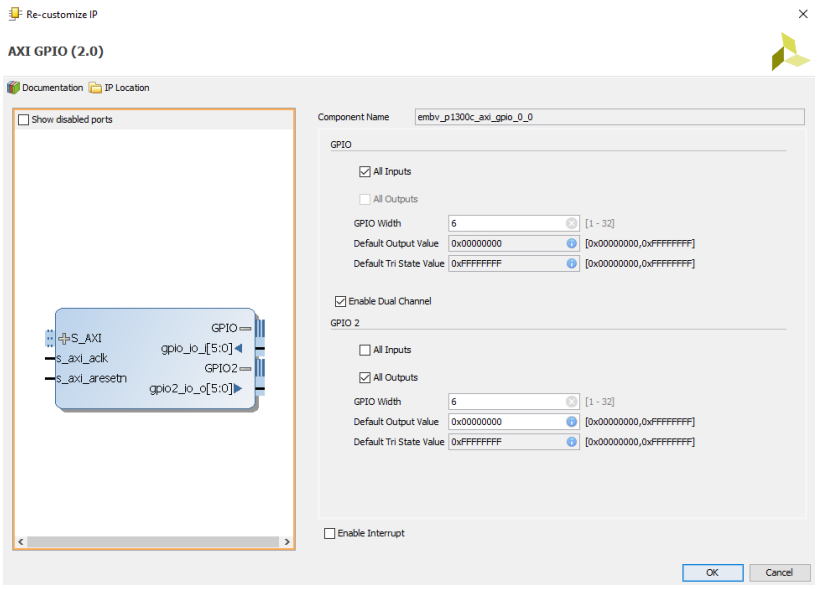


Figura 6.6: Configuración bloque AXI GPIO

El bus de entrada es conectado al puerto *S2MM_FRAME_PTR_OUT[5:0]* del bloque VDMA que indica cual es el último *buffer* completamente escrito por el canal de escritura y, por tanto, donde se almacena el *frame* más reciente capturado por la cámara. El bus de salida es conectado al bloque que concatena los dos buses para posteriormente conectar este bus resultante al puerto *S2MM_FRAME_PTR_IN[11:0]* del bloque VDMA.

En la Figura 6.7 se puede observar en detalle los bloques y sus conexiones explicados en este apartado para controlar el acceso a los *buffers* almacenados en memoria DDR.

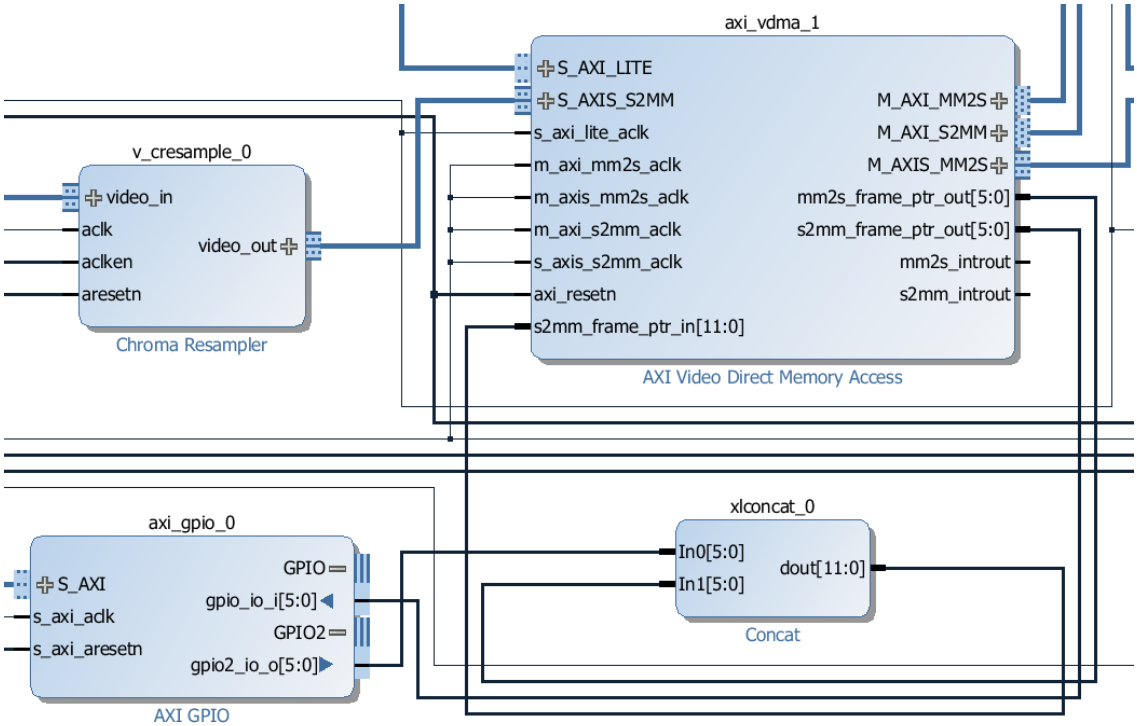


Figura 6.7: Conexionado para el control del acceso a los buffers en DDR

Gracias a este diseño, cada vez que una imagen comienza a ser procesada por el bloque HOG, el *buffer* donde esta está almacenada es *bloqueado* de forma que no pueda ser sobrescrito

por el canal de escritura del bloque VDMA. Una vez que este *buffer* ha sido completamente procesado, se *desbloquea* este y se comprueba cual es el *buffer* más recientemente escrito. Este será posteriormente bloqueado para proceder a su procesado por el bloque HOG.

Cabe destacar que el resto de conexiones de sincronización internas del bloque VDMA siguen siendo utilizadas por el canal de lectura para conocer en todo momento cual es el *buffer* con el *frame* más reciente por lo que, en relación al canal de lectura, la sincronización sigue llevándose a cabo de forma interna.

Una vez realizadas las conexiones anteriores, se puede proceder a realizar la síntesis e implementación del diseño. En la Figura 6.8 se puede observar el uso de recursos utilizados por este diseño.

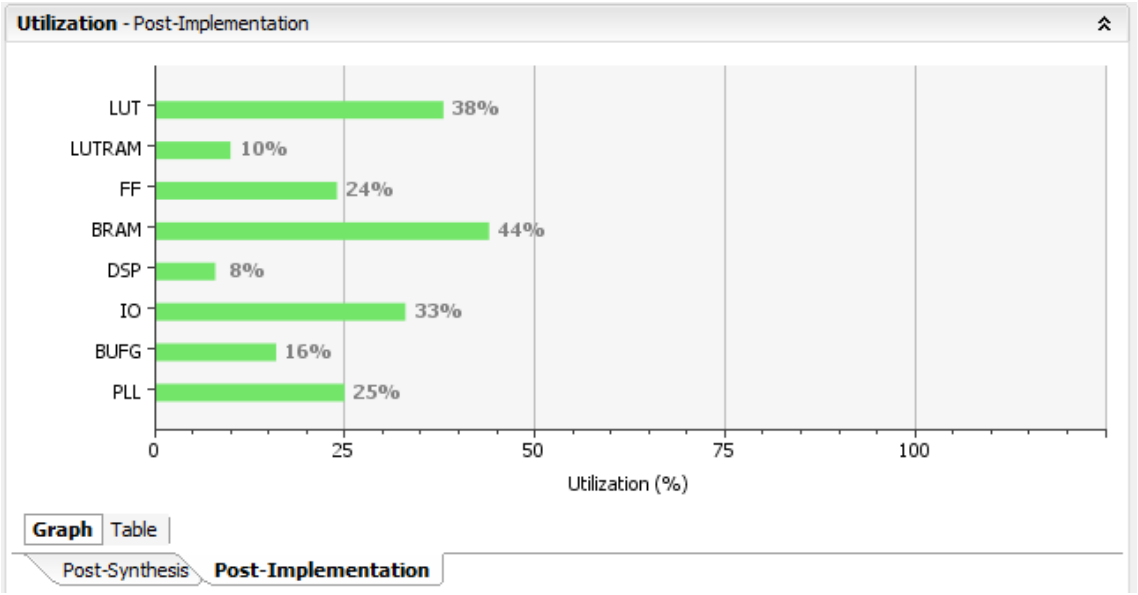


Figura 6.8: Recursos HW utilizados para la implementación SW del detector SVM

Por último, se procede a generar y exportar el archivo *bitstream* que se utilizará para programar el *hardware* del dispositivo Zynq.

6.3.2 Aplicación en Vivado SDK

La aplicación *software* de este diseño se encargará, además de realizar toda la configuración del sistema de captura y visualización de imágenes ya explicada detalladamente en el capítulo 3, de controlar la ejecución del bloque HOG. Además, a partir de los resultados de este y de los pesos SVM, se decidirá sobre la detección o no de persona en cada una de las ventanas procesadas.

El objetivo de la aplicación es la localización de personas en la imagen capturada por la cámara. Por ello, es importante recorrer esta imagen de forma detallada, moviendo la ventana de detección con desplazamientos muy reducidos y, por tanto, superponiendo las ventanas entre sí. En el caso de esta aplicación, se ha configurado un desplazamiento de ventana de 8 píxeles en sentido horizontal y de 16 píxeles en sentido vertical. Esto supone que, para una imagen de 1280x1024, el número de ventanas a procesar sea de 8512.

6.3.2.1 *Pseudocódigo* de la aplicación

Con el objetivo de facilitar la comprensión de la aplicación desarrollada, se incluye en este apartado un *pseudocódigo* de esta.

```
int main()
{
	//Inicializaciones y configuraciones del sistema de captura y
visualización de imágenes (explicado detalladamente en el capítulo 3
de este documento)
	//Definición de constantes: vector pesos SVM
	//Inicialización bloque IP HOG simplificado y bloque AXI GPIO
	while(1)
	{
		//Se lee el último buffer escrito por el bloque VDMA y se
bloquea
		//Configuración del bloque HOG para leer el buffer
bloqueado
		//Se recorren las diferentes ventanas que se desean
procesar en una imagen
		for(y=0; y<num_ventanas_y; y++) //Para cada ventana en
dirección vertical
		{
			for(x=0; x<num_ventanas_x; x++) //Para cada ventana
en dirección horizontal
			{
				//Cálculo de la ubicación en el vector imagen
del píxel de la esquina superior izquierda de la ventana a procesar
				//Configuración del bloque IP HOG simplificado
				//Inicio ejecución bloque HOG en HW
				//Espera hasta la finalización de la ejecución
del bloque HOG en HW
				//Decisión sobre la detección de persona en la
ventana analizada empleando el descriptor HOG calculado y el vector de
pesos de la SVM
				//Si se determina la existencia de persona,
configuración del bloque OSD para dibujar un rectángulo de 64x128
píxeles sobre la imagen mostrada en el monitor
			}
		}
	}
}
```

Como se ha comentado anteriormente, la configuración de los distintos bloques que componen el sistema de captura y visualización de imágenes se ha explicado detalladamente en el apartado 3.2 de esta memoria, por lo no se va a volver a detallar en este apartado.

Sin embargo, es necesario modificar ligeramente la configuración del bloque VDMA de forma que se le indique que las señales de sincronización del canal de escritura son configuradas de forma externa, debiendo por tanto obviar las señales internas. Para ello, se debe ejecutar la siguiente función entre el *reset* del canal y su configuración.

```
XAxiVdma_GenLockSourceSelect(pdemo->paxivdma0,
XAXIVDMA_EXTERNAL_GENLOCK,XAXIVDMA_WRITE);
```

En el apartado 5.7.2 del capítulo anterior se ha detallado como se va a recorrer la imagen y como debe ser configurado el bloque HOG de manera que se procesen las distintas ventanas de forma correcta. Debido a que dicha configuración es muy similar a la necesaria para esta aplicación, únicamente se detallarán en este apartado las modificaciones efectuadas.

En el caso de esta aplicación, es necesario modificar las configuraciones relativas a las direcciones de memoria DDR desde donde se leerán las ventanas de detección y en la que se escribirá el descriptor tras su procesamiento.

En relación a la dirección del bus AXI de lectura del bloque HOG, esta se configurará cada vez que se empiece a procesar una imagen y dependerá de la información leída a través del bloque AXI GPIO.

```
buffer=XGpio_DiscreteRead(&gpio, 1); //Lectura del identificador del
buffer más recientemente accedido por el canal de escritura del bloque
VDMA
XGpio_DiscreteWrite(&gpio, 2, buffer); //Escritura del identificador
del buffer que se va a procesar por el bloque HOG
```

Tal y como se comentó en el apartado 3.1.2.7.2.3 la identificación de los puertos sigue un código *Gray* circular y duplicado. Por ello, se ha utilizado el siguiente código para determinar en qué dirección comienza el *buffer* que se va a procesar.

```
switch (buffer)
{
      case 0:
            dir=0x18000000;
            break;
      case 1:
            dir=0x18280000;
            break;
      case 3:
            dir=0x18500000;
            break;
      case 2:
            dir=0x18780000;
            break;
      case 6:
            dir=0x18000000;
            break;
      case 7:
            dir=0x18280000;
            break;
      case 5:
            dir=0x18500000;
            break;
      case 4:
            dir=0x18780000;
            break;
}
```

Una vez determinada la dirección de la cual se debe leer de memoria, esta debe ser configurada en el bloque HOG con la siguiente función:

```
XHog_Set_imagen(&HOG0, dir);
```

En relación a la interfaz AXI a través de la cual se escriben los descriptores calculados, en esta aplicación no es necesario almacenar todos los obtenidos en una imagen, sino que cada vez que uno de ellos sea calculado, se decidirá sobre la existencia o no de persona en la ventana asociada. Por ello, la dirección de escritura se mantendrá fija en todo momento, no

siendo necesario llamar a esta función más que en una ocasión al principio de la ejecución de la aplicación.

```
XHog_Set_descriptor_hog(&HOG0,descriptor_hog);
```

6.3.2.2 Decisión de clasificación empleando SVM

Cada vez que el algoritmo HOG termina de ejecutarse, una ventana ha sido procesada y su descriptor asociado se encuentra almacenado en memoria DDR a partir de la dirección apuntada por *descriptor_hog*. Empleando la función discriminante obtenida tras el entrenamiento de la SVM, se decidirá sobre la detección o no de persona en la ventana procesada. Para ello, se inicializa una variable con el valor del término independiente obtenido tras el entrenamiento de la SVM y se acumula en ella el producto de cada uno de los elementos del descriptor con sus valores correspondientes del vector de pesos.

Cabe destacar que los elementos del descriptor HOG calculados por el bloque *hardware* se encuentran almacenados en memoria en formato coma fija con un tamaño de palabra de 16 *bits*, dejando para la parte decimal 15 de esos *bits*. Por ello, antes de realizar las distintas operaciones, se debe dividir cada valor entre 2^{15} de forma que se obtenga el valor equivalente en formato flotante.

El código utilizado para obtener el valor que permita decidir si en una ventana se localiza una persona o no se muestra a continuación:

```
suma = -3.4869;
for(ind=0; ind<1152; ind++)
{
      temporal=descriptor_hog[ind]/(float)(pow(2,15));//Lectura y
conversión a formato flotante
      suma=suma+pesos_svm[ind]*temporal;//Aplicación del discriminante
}
```

De forma general, si el valor de *suma* es mayor que 0, se decidiría la existencia de una persona en la ventana. Por el contrario, si este valor fuera negativo, se clasificaría la ventana como *no persona*. Sin embargo, con el objetivo de evitar en la medida de lo posible la aparición de falsos positivos, se va a umbralizar este valor, de forma que únicamente se clasificará como *persona* cuando *suma* sea superior a este umbral. En el caso de esta aplicación, se ha determinado emplear un umbral de 1,5 para clasificar una ventana como *persona*.

6.3.2.3 Empleo del bloque OSD para la señalización de personas

Una vez detectada una persona, esta va a ser señalada en la imagen mostrada en el monitor mediante un rectángulo. Para ello, se va a hacer uso del bloque OSD y del sencillo controlador gráfico que este alberga.

En este capítulo se ha añadido una segunda capa que, al igual que la primera, debe ser configurada y habilitada al inicio de la ejecución con las siguientes funciones:

```
XOSD_SetLayerAlpha(pdemo->posd, 1, 0, 0xFF); //Configuración del
parámetro alpha
XOSD_SetLayerDimension(pdemo->posd, 1, 0, 0, 1280, 1024);
//Configuración del tamaño de la capa
XOSD_EnableLayer(pdemo->posd, 1); //Habilitación de la capa
```

Para poder dibujar un rectángulo sobre la ventana en la que se ha detectado una persona, el primer paso es conocer la posición de esta en la imagen completa. Para ello, se emplean

las siguientes expresiones que tienen en cuenta el número de ventanas procesadas en dirección horizontal y vertical, *x* e *y*, los desplazamientos en cada dirección y el propio tamaño de la ventana.

```
x1 = (x*stridex); //Primera columna de la ventana en la imagen
x2 = (x*stridex+64); //Última columna de la ventana en la imagen
y1 = (y*stridey); //Primera fila de la ventana en la imagen
y2 = (y*stridey+128); //Última fila de la ventana en la imagen
```

El controlador gráfico de este bloque genera las formas configuradas en una lista de *instrucciones* que recogen las características de estas. Las instrucciones son generadas usando la función *XOSD_CreateInstruction* que permite configurar estas según sean de tipo caja o texto.

Para configurar la representación de un rectángulo los argumentos que la función *XOSD_CreateInstruction* necesita son los siguientes:

```
XOSD_CreateInstruction(pdemo->posd, Instruction[num_ins], 1,
XOSD_INS_OPCODE_BOX, ObjSize, x1, y1, x2, y2, 0, ColorIndex);
```

La constante *XOSD_INS_OPCODE_BOX* indica que se desea dibujar un rectángulo cuyos límites vienen dados por *x1*, *y1*, *x2* e *y2* y el grosor de la línea y su color por las variables *ObjSize* y *ColorIndex*.

Una vez que se ha terminado de procesar una imagen completa, se representan sobre la imagen todos los rectángulos configurados empleando la función *XOSD_LoadInstructionList* que recibe como parámetros el conjunto de instrucciones generadas y su número.

```
XOSD_LoadInstructionList(pdemo->posd, 1, 0, Instruction, num_ins);
```

6.3.3 Ejecución de la aplicación y resultados

Una vez se ha realizado el diseño completo de la aplicación, esta puede ser ejecutada empleando los mismos dispositivos y conexionado que en el capítulo 3.

Un ejemplo de los resultados obtenidos tras la ejecución del diseño se puede observar en la Figura 6.9.

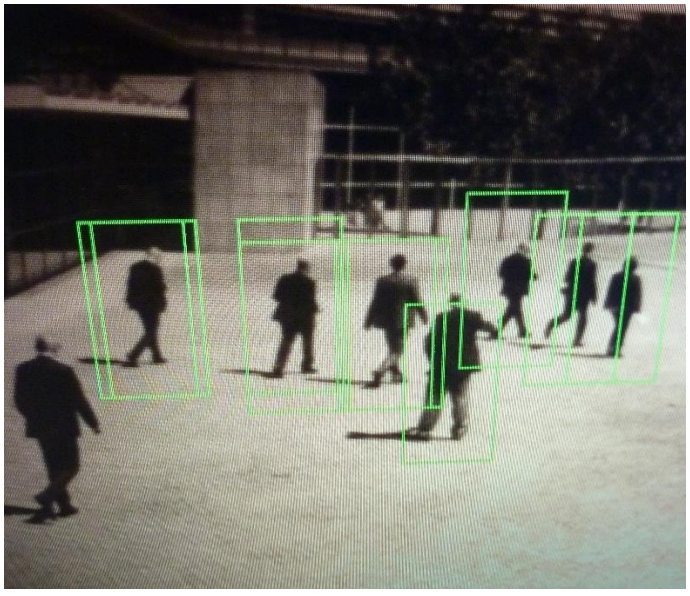


Figura 6.9: Resultados de la ejecución de la aplicación HOG + SVM

En relación a los tiempos de ejecución, se obtiene que cada ventana tarda en ser procesada 1,4 ms, lo que supone que el procesado de 8512 ventanas de una imagen necesita aproximadamente 12 segundos.

Se puede observar que, restando el tiempo necesario para calcular los descriptores HOG en HW, 700 µs, el tiempo obtenido para aplicar la función discriminante de la SVM en *software* es también 700 µs. Este tiempo se puede considerar bastante elevado por lo que, para intentar reducirlo, se va a incluir la función discriminante de la SVM dentro del bloque *hardware* encargado del cálculo del algoritmo HOG simplificado.

6.4 Implementación HW del detector SVM

En el apartado anterior se ha aplicado en *software* la función discriminante de la máquina SVM a los descriptores calculados por el bloque HW. Sin embargo, esta función puede ser añadida al bloque HOG implementado en HW de forma relativamente sencilla, permitiendo acelerar la ejecución del conjunto.

Para ello, se ha partido del código C diseñado en HLS para el cálculo del algoritmo HOG y se ha modificado ligeramente, añadiendo el cálculo de la función discriminante de la SVM entrenada. Una vez sintetizado y generado el bloque HW, se ha añadido al proyecto Vivado y se han realizado las modificaciones necesarias en la aplicación *software*.

6.4.1 Modificación de la función HLS

El diseño HLS anterior se encargaba de escribir en memoria DDR los descriptores calculados tras el procesamiento de una ventana. Sin embargo, para esta aplicación, estos descriptores son empleados por el propio bloque, multiplicando y acumulando cada uno de sus elementos por los pesos de la SVM entrenada. Así, el resultado esperado para este bloque es un único valor que puede ser accedido desde la aplicación *software* empleando la interfaz *AXI-Lite* ya disponible.

De esta forma, la definición de la función *top* se ha modificado, quedando tal y como se muestra a continuación:

```
void hog(int inicio_ventana, volatile unsigned char *imagen, float
*resultado_svm)
```

Se puede observar que la función tiene como argumento, además de los ya explicados anteriormente, un puntero de tipo flotante en el cual se almacenará el resultado obtenido tras finalizar el procesado. Cabe destacar que se ha empleado una variable de tipo flotante para simplificar la conversión posterior cuando este sea leído desde *software*.

Para indicar a la herramienta HLS que se desea que este argumento de la función sea implementado como una interfaz *AXI-Lite* es necesario emplear el *pragma interface*, tal y como se muestra a continuación:

```
#pragma HLS INTERFACE s_axilite port=resultado_svm bundle=AXI_Lite
```

Una vez se han realizado los cambios relativos a las interfaces de comunicación, únicamente queda añadir el cálculo de la función discriminante de la máquina vector soporte.

Para ello, el primer paso es definir como constante el vector de 1152 pesos de la SVM entrenada. Con el objetivo de optimizar el código, estos pesos se declaran empleando una notación en coma fija con un tamaño de palabra de 18 *bits* de los cuales, 4 serán para la parte entera.

```
const ap_fixed<18, 4> pesos_svm[1152]={0.120984, -0.327067, …};
```

La función discriminante consiste en la acumulación del producto de cada intervalo normalizado por el valor asociado del vector de pesos. Cabe destacar que la variable *suma* es inicializada al comienzo de la ejecución de una ventana con el valor *bias* obtenido en Matlab tras el entrenamiento de la SVM.

Con el objetivo de que esta operación sea realizada de forma concurrente para varios intervalos, se ha utilizado el *pragma UNROLL*. Esta directiva *desenrolla* un bucle ejecutando de forma paralela el número de iteraciones indicado en el parámetro *factor*. De esta forma se consigue reducir el número de ciclos necesarios para la ejecución pero se incrementa el número de recursos utilizados.

```
for (int m = 0; m < 36; m++)
{
#pragma HLS UNROLL factor=8
      temp2=histogramas[m]*inversa; //Normalización del intervalo del
histograma
      suma = suma + temp2*pesos_svm[j*36+m]; //Aplicación de la
función discriminante
      histogramas[m]=0; //Inicialización del intervalo del histograma
}
```

Una vez que se han procesado todos los bloques de una ventana, se escribe en el registro del bus *AXI-Lite* correspondiente el valor obtenido, convirtiéndolo previamente a un formato en coma flotante. Como sea ha comentado anteriormente, esta conversión se lleva a cabo únicamente para simplificar su tratamiento posterior en la aplicación *software*.

```
(*resultado_svm)=(float)suma;
```

Realizadas las modificaciones anteriores, se procede a la síntesis de la función, que genera una estimación temporal y de recursos mostrada en la Figura 6.10. Se puede observar que, comparado con la estimación tras la síntesis del bloque HOG mostrada en la Figura 5.18, el uso de DSPs ha aumentado substancialmente, manteniéndose prácticamente igual el uso del resto de recursos. Esto es debido a que estos dispositivos están especializados en las operaciones matemáticas de multiplicación y acumulación [17], que se corresponden exactamente a las necesidades del cálculo de la función discriminante.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	6.67	6.38	0.83

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
34602	34602	34603	34603	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	8	-	-
Expression	-	-	0	4349
FIFO	-	-	-	-
Instance	0	16	1392	1803
Memory	42	-	304	81
Multiplexer	-	-	-	2179
Register	-	-	3589	32
Total	42	24	5285	8444
Available	280	220	106400	53200
Utilization (%)	15	10	4	15

Figura 6.10: Informe HLS tras proceso de síntesis

Por último, se procede a cosimular y exportar el bloque IP para que pueda ser empleado en la siguiente fase del diseño de la aplicación.

6.4.2 Modificación del diseño en Vivado y Vivado SDK

Una vez se ha generado el bloque IP que calcula el algoritmo HOG simplificado y la función discriminante de la SVM, este debe ser añadido al proyecto en Vivado, sustituyendo al bloque HOG instanciado en el apartado 6.3.

En la Figura 6.11 se puede observar un fragmento del diseño en Vivado tras sustituir el bloque HW.

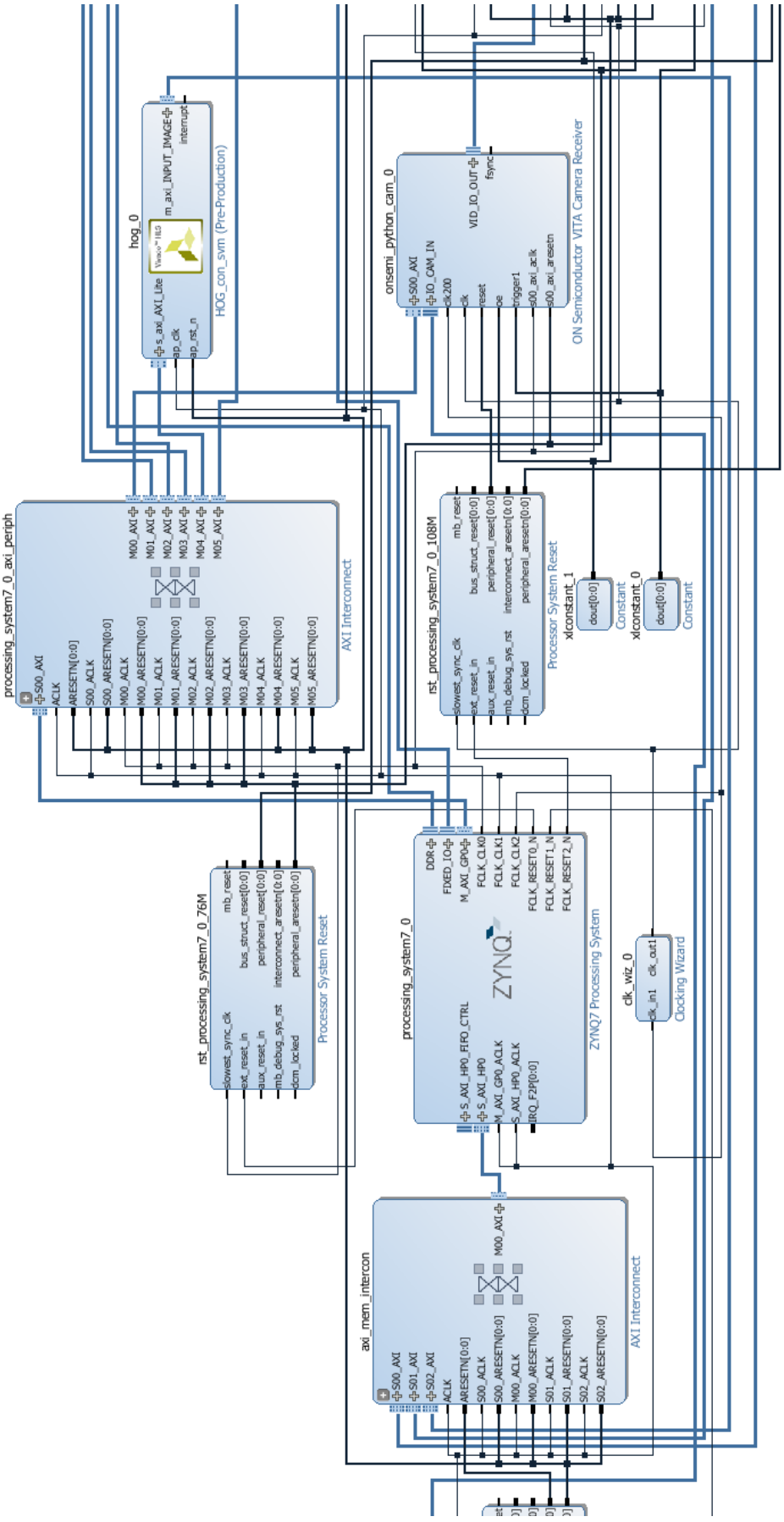


Figura 6.11: Fragmento diseño en Vivado con bloque IP HOG + SVM

En relación al diseño *software*, este únicamente debe ser modificado eliminando las expresiones relacionadas con la aplicación de la función discriminante y añadiendo la lectura del resultado devuelto por el bloque *hardware*. Esta lectura se realiza empleando la función *XHog_Get_resultado_svm* que devuelve el valor del registro asociado a la variable *resultado_svm* pasada como argumento de la función *top* en HLS. Sin embargo, el valor leído se encuentra en un formato *int32* que, antes de poder ser comparado con el umbral de detección, debe ser convertido a un formato decimal.

```
suma_int=XHog_Get_resultado_svm(&HOG0); //Lectura del valor calculado
por la función HW
suma =*((float*)&suma_int); //Conversión a formato float
```

Tal y como se ha explicado en el apartado 6.3.2, este valor es comparado con el umbral fijado para determinar la detección de persona y, si es superado, se marca su posición por medio de un rectángulo generado por el bloque OSD.

6.4.3 Resultados obtenidos

Tras la ejecución de la aplicación, se ha obtenido un tiempo de procesado por ventana de 650 µs, lo que supone aproximadamente 5,3 segundos para clasificar las 8512 ventanas procesadas para una imagen completa, representando una aceleración de aproximadamente el doble con respecto al SVM en SW.

Cabe destacar que este tiempo es ligeramente inferior al obtenido por el bloque HOG sin el cálculo de la función discriminante SVM. Esto puede ser explicado por el hecho de que este bloque no realiza múltiples escrituras a memoria DDR, sino que únicamente devuelve un valor por cada ventana procesada.

De esta forma, el tiempo que se tarda en realizar las 1152 escrituras a memoria tras el cálculo de la función simplificada HOG es compensado por el cálculo de las 1152 operaciones de multiplicación y acumulación de la función discriminante SVM.

6.5 Alternativas de diseño según especificación para la detección de personas en una imagen

Hasta este momento, se ha considerado que la detección de personas se debía realizar en toda la imagen y con una precisión elevada, superponiendo las ventanas entre sí. Esto supone una gran cantidad de ventanas a procesar y, por tanto, un tiempo de procesamiento elevado. Para intentar mejorar estos aspectos, se proponen dos métodos que pueden ayudar a que cada imagen sea procesada más rápidamente.

6.5.1 Paralelización del cálculo HOG + SVM

Hasta este momento, el cálculo de las diferentes ventanas que se calculan para cada *frame* se realiza de forma secuencial, una detrás de otra. Sin embargo, este proceso puede ser fácilmente paralelizable instanciando varios bloques IP para el cálculo del algoritmo HOG en el diseño en Vivado.

Se han realizado diferentes implementaciones, llegando a acumular hasta 4 bloques HOG + SVM en paralelo. Sin embargo, se ha comprobado que la velocidad de procesamiento de una imagen entera se ve limitada por los accesos a memoria DDR de forma concurrente por lo que los tiempos apenas varían cuando se emplean más de 2 bloques IP en paralelo. Para esta configuración, los tiempos obtenidos en el procesamiento de una imagen entera, procesando por tanto 8512 ventanas, es de aproximadamente 3 segundos.

En la Figura 6.12, se puede observar un ejemplo de los resultados obtenidos cuando se paraleliza el cálculo del algoritmo HOG. Para una mejor observación de los resultados, se emplea un color diferente para marcar las personas detectadas, dependiendo del bloque HW que se haya encargado del procesamiento.

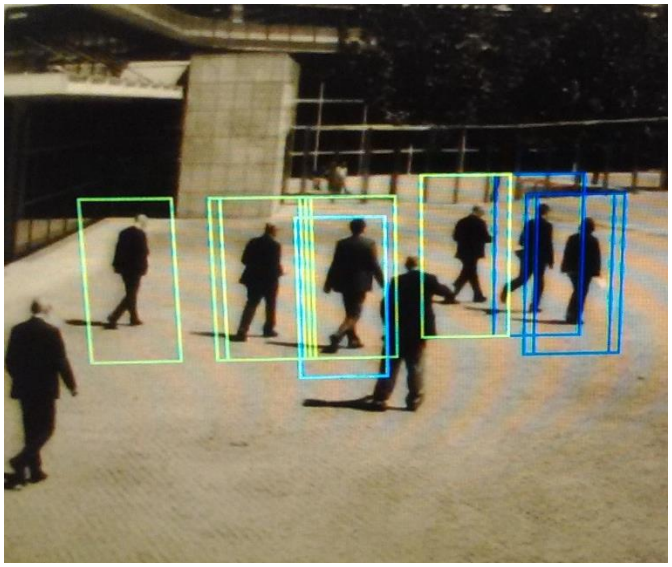


Figura 6.12: Resultados de la ejecución de la aplicación HOG + SVM con paralelización

6.5.2 Selección de una región de interés

Hasta este momento se ha considerado que las personas podían aparecer en cualquier zona de la imagen. Sin embargo, es posible conocer de antemano que en ciertas zonas de la imagen estas no van a aparecer nunca. A modo de ejemplo, si la cámara está enfocada a una calle y secciones de la imagen corresponden a edificios, es posible obviar estas zonas y procesar únicamente las partes que nos interesan.

De esta forma, el número de ventanas procesadas por imagen, suponiendo un solapamiento fijo, se reduce, permitiendo procesar más rápidamente cada *frame*. A modo de ejemplo, en este caso se ha supuesto que la región de interés corresponderá con la zona central de la imagen, equivalente a un cuarto de la imagen. En este caso, se ha obtenido un tiempo de procesado por imagen de aproximadamente 800 milisegundos.

Capítulo 7: Conclusiones y líneas futuras

En este capítulo se detallan las conclusiones obtenidas a lo largo de este trabajo y se indican las posibles líneas futuras de investigación.

7.1 Conclusiones

En relación a los resultados obtenidos, se enumeran las siguientes conclusiones:

- Se ha realizado el análisis, conexión y puesta en marcha de las distintas tarjetas que forman el sistema de procesamiento de video en HW, indicando sus principales características y componentes.
- Se ha implementado un sistema de captura, procesamiento y visualización de imágenes en tiempo real sobre un dispositivo *SoC*. Para ello, se ha detallado el diseño *hardware* empleado, explicando las diferentes opciones de configuración de cada uno de los bloques HW utilizados. Además, se ha comentado la aplicación *software* necesaria para la correcta configuración de los distintos componentes que conforman el diseño. Por último, se ha comprobado el correcto funcionamiento del sistema completo.
- Se ha desarrollado y añadido al diseño anterior un algoritmo HW/SW que calcula el histograma de las imágenes capturadas y lo muestra superpuesto a estas en tiempo real.
- Se ha utilizado de forma intensiva la herramienta Vivado HLS para implementar en *hardware* diferentes algoritmos SW, teniendo que aprender y usar las directivas que controlan dicha implementación.
- Se ha estudiado detalladamente el algoritmo HOG, indicando su funcionamiento y sus principales parámetros y configuraciones. Se ha evaluado el algoritmo original tanto en Matlab como en código C en SW. Se ha justificado e implementado una simplificación del algoritmo HOG original.
- Se ha realizado la implementación *hardware* del algoritmo HOG simplificado, analizando las limitaciones que este impone. Se han comparado los resultados de ejecución del módulo IP HW con los obtenidos al ejecutar el mismo algoritmo en *software*. Se ha desarrollado una aplicación completa para la evaluación del correcto funcionamiento de ambos diseños, tanto HW como SW del algoritmo HOG.
- Por último, se ha procedido al diseño de una aplicación para la detección de personas en imágenes. Para ello, se ha empleado un algoritmo de aprendizaje supervisado SVM utilizando como observaciones los descriptores obtenidos por el módulo HW de HOG. Se ha realizado el entrenamiento de dicho sistema SVM con imágenes de un *dataset* público y su función discriminante se ha implementado tanto en *software* como en *hardware*, comparando sus resultados finalmente.

Para cada una de las tareas que se han descrito anteriormente, a lo largo del presente TFM se han ido proporcionando resultados de la implementación de algoritmos en SW y HW.

De la lectura de la memoria se puede extraer como conclusión que la implementación de un algoritmo en HW a través de un sistema de descripción *hardware* en alto nivel como HLS necesita que el diseñador tenga ciertos conocimientos de electrónica digital. La implementación HW generada depende en gran medida del buen empleo que se haga de las directivas que rigen la generación del sistema HW y las líneas de código que definen el algoritmo en SW.

En relación al rendimiento obtenido, se ha podido comprobar que el cuello de botella principal para que el rendimiento no sea mayor es el acceso a memoria DDR. Por ese motivo, en este tipo de sistemas, se debe hacer un esfuerzo adicional en la mejora de la eficiencia en el acceso a los datos, utilizando dobles *buffers*, bloques DMA con copia/lectura de datos, etc.

Por último, en la Figura 7.1 se muestran diversas capturas del sistema empotrado junto con los resultados obtenidos para las diversas aplicaciones desarrolladas en este trabajo. Aunque este sistema se ha conectado a un monitor HDMI para visualizar las imágenes en tiempo real y a un ordenador para su control, tiene un funcionamiento autónomo y puede ser instalado en una ubicación remota formando, por ejemplo, parte de un sistema IoT.

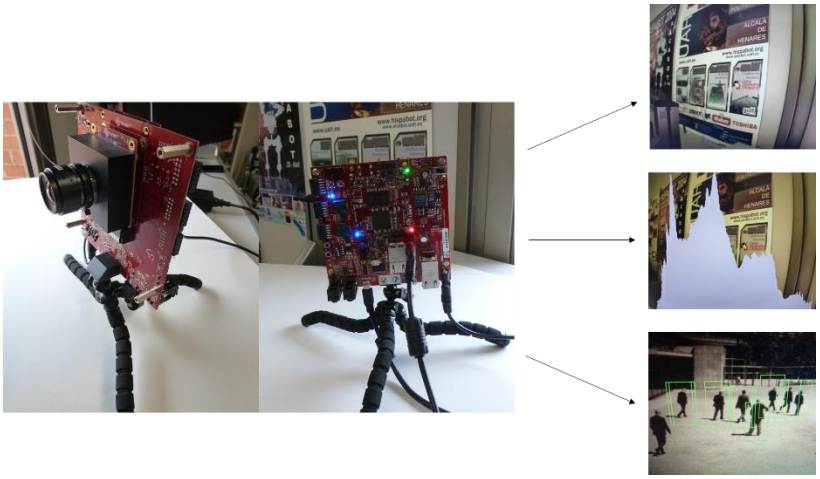


Figura 7.1: Sistema SoC en funcionamiento para procesamiento de video en tiempo real

7.2 Líneas futuras

En este apartado se indican algunas de las posibles líneas por las que se podría continuar el desarrollo del trabajo:

- Con el objetivo de disminuir los tiempos de procesamiento, se propone paralelizar el cálculo del algoritmo HOG de cada una de las ventanas, dividiendo esta en regiones más pequeñas que sean calculadas de forma concurrente.
- Implementación del algoritmo directamente sobre el flujo de vídeo, evitando así los accesos a memoria DDR que ralentizan el procesado.
- Con el objetivo de poder detectar personas que tengan diferentes tamaños en la imagen, se propone realizar múltiples escalados de la imagen, permitiendo así ajustar el tamaño de las personas al tamaño de la ventana HOG.
- Empleo de algoritmos de seguimiento de las personas detectadas, como el filtro de Kalman o el filtro de partículas, que permiten reducir el número de ventanas a procesar por cada imagen.

Pliego de condiciones

Los equipos, dispositivos y herramientas *software* empleadas en este trabajo se detallan a continuación.

Recursos hardware

- Ordenador utilizado en el desarrollo
  - Procesador Intel Core i5-7500 de 3,4 GHz
  - 32 GB de memoria RAM
  - Disco duro SSD de 256 GB de memoria
- Tarjeta de desarrollo MicroZed Z020
- MicroZed Embedded Vision Carrier Card
- ON Semiconductor PYTHON 1300-C Camera Module
- Cable de programación/desarrollo JTAG
- Monitor HDMI
- Cables: HDMI a Micro-HDMI, 2xUSB a USB a Micro-USB, Ethernet

Recursos software

- Sistema operativo Windows 10 de 64 bits
- Xilinx Vivado Design Suite 2015.4 o superior
- MATLAB 2018 o superior
- TeraTerm
- Suite Microsoft Office

Presupuesto

En este capítulo se recogen los distintos costes del trabajo realizado. Los costes pueden ser divididos en dos tipos: coste directo (CD), que engloba el coste del material y de la mano de obra, y el coste indirecto (CI), que recoge los gastos generales.

En la siguiente tabla se incluyen los gastos referentes al material y al *software* utilizado, considerados como CD. Cabe destacar que los equipos electrónicos tienen una amortización estimada de 4 años, por lo que se debe estimar el coste en función del tiempo que han sido utilizados.

Desglose de costes en términos de material			
Material/*Software*	**Precio (euros)**	**Utilización/Amortización**	**Coste al proyecto (euros)**
Ordenador	1000 €	8 meses/4 años	166,67 €
MicroZed	332 €	8 meses	332 €
MicroZed Embedded Vision Carrier Card	386 €	8 meses	386 €
ON Semiconductor PYTHON 1300-C Camera Module	502 €	8 meses	502 €
Cable JTAG	47 €	8 meses	47 €
Monitor HDMI	170 €	8 meses/4 años	28,33 €
Xilinx Vivado Design Suite	2995 €	8 meses	2995 €
Licencia MATLAB	35 €	8 meses	35 €
Suite Microsoft Office	149 €	8 meses	149 €

El coste total de material y *software* utilizado para la realización de este trabajo es de 4641 euros sin incluir impuestos.

En la siguiente tabla se muestra el coste por mano de obra del trabajo, indicando por separado el precio por horas de ingeniería y de redacción del trabajo. Al igual que el anterior, este coste se considera directo.

Desglose de costes en términos del trabajo realizado			
Trabajo	**Número de horas**	**Precio por hora (euros)**	**Total (euros)**
Ingeniero	400	60 €/hora	24000 €
Redacción	100	20 €/hora	2000 €

El coste total por la mano de obra, suma del coste por el trabajo de ingeniería y de redacción, es de 26000 euros sin incluir impuestos.

El total de los costes directos (CD) se corresponde con la suma de los costes del material (4641 euros) y de la mano de obra (26000 euros), por lo que los costes directos del trabajo son 30641 euros.

A los costes anteriores, hay que sumarle los gastos generales, considerados costes indirectos (CI) que se calculan como el 15% del CD. Además, hay que sumar el beneficio industrial que se calcula como el 6% de la suma del CD más el CI. En la siguiente tabla se muestran estos costes desglasados.

Desglose de costes del resto de conceptos			
Concepto	**Porcentaje**	**Coste aplicado**	**Total (euros)**
Gastos Generales (CI)	15%	CD	4596,15 €
Beneficio Industrial	6%	CD + CI	2114,23 €

El precio, una vez añadido el IVA, se muestra en la siguiente tabla.

Costes del proyecto con IVA		
Precio sin IVA (euros)	**IVA**	**Total (euros)**
37351,38 €	21%	45195,17 €

El coste total del proyecto con el IVA ya incluido es de 45195,17 euros, CUARENTA Y CINCO MIL CIENTO NOVENTA Y CINCO CON DIECISIETE.

Bibliografía

[1] Xilinx, «Zynq-7000 SoC Technical Reference Manual,» [En línea]. Disponible: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

[2] Avnet, «MicroZed™ Zynq® Evaluation Kit and System on Module Hardware User Guide,» [En línea]. Disponible: http://microzed.org/sites/default/files/documentations/5276-MicroZed-HW-UG-v1-7-V1.pdf.

[3] Avnet, «MicroZed™ Embedded Vision Carrier Card Hardware User Guide,» [En línea]. Disponible: http://microzed.org/sites/default/files/documentations/5188-UG-AES-MBCC-EMBV-DEV-G-V1.pdf.

[4] Analog Devices, «ADV7511 Hardware User's Guide,» [En línea]. Disponible: https://www.analog.com/media/en/technical-documentation/user-guides/ADV7511_Hardware_Users_Guide.pdf.

[5] Analog Devices, «ADV7511 Programming Guide,» [En línea]. Disponible: https://www.analog.com/media/en/technical-documentation/user-guides/ADV7511_Programming_Guide.pdf.

[6] Avnet, «ON Semiconductor PYTHON 1300-C Camera Module Hardware User Guide,» [En línea]. Disponible: http://microzed.org/sites/default/files/documentations/BD-CAM-PYTHON1300C-B%20User%20Guide%20-%20v1.0-1.pdf.

[7] Avnet, «Avnet Github,» [En línea]. Disponible: https://github.com/Avnet/hdl.

[8] ON Semiconductor, «PYTHON 1.3/0.5/0.3 MegaPixels Global Shutter CMOS Image Sensors,» [En línea]. Disponible: https://www.onsemi.com/pub/Collateral/NOIP1SN1300A-D.PDF.

[9] F. Bensaali y A. Amira, «Design and implementation of efficient architectures for color space conversion,» *ICGST International Journal on Graphics, Vision and Image Processing,* vol. 5, nº 1, pp. 37--47, 2004.

[10] Xilinx, «Chroma Resampler v4.0 LogiCore IP Product Guide,» [En línea]. Disponible: https://www.xilinx.com/support/documentation/ip_documentation/v_cresample/v4_0/pg012_v_cresample.pdf.

[11] Xilinx, «AXI Video Direct Memory Access v6.2 LogiCore IP Product Guide,» [En línea]. Disponible: https://www.xilinx.com/support/documentation/ip_documentation/axi_vdma/v6_2/pg020_axi_vdma.pdf.

[12] N. Dalal y B. Triggs, «Histograms of Oriented Gradients for Human Detection,» de *International Conference on Computer Vision & Pattern Recognition (CVPR '05)}*, San Diego, United States, 2005.

[13] N. Dalal, Finding People in Images and Videos, 2006.

[14] B. E. Boser, I. M. Guyon y V. N. Vapnik, «A training algorithm for optimal margin classifiers,» de *Proceedings of the fifth annual workshop on Computational learning theory*, 1992.

[15] N. Dalal, «INRIA Person Dataset,» [En línea]. Disponible: http://pascal.inrialpes.fr/data/human/.

[16] Google Developers, «Aprendizaje automático - Clasificación,» [En línea]. Disponible: https://developers.google.com/machine-learning/crash-course/classification/video-lecture.

[17] Xilinx, «7 Series DSP48E1,» [En línea]. Disponible: https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf.

Universidad de Alcalá

Escuela Politécnica Superior

ESCUELA POLITECNICA SUPERIOR