ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Proyecto Fin de Carrera

IMPLEMENTACIÓN HARDWARE DE DETECTOR DE OJOS SOBRE VIDEO EN TIEMPO REAL

(Hardware Implementation of a Real Time Eye Detection in Video Signals)

Para acceder al Título de

INGENIERO DE TELECOMUNICACIÓN

Autor: Roberto Ibáñez Zabala

Septiembre - 2017

INGENIERÍA DE TELECOMUNICACIÓN

CALIFICACIÓN DEL PROYECTO FIN DE CARRERA

Realizado por: Roberto Ibáñez Zabala
Director del PFC: Pablo Pedro Sánchez Espeso
Título: "IMPLEMENTACIÓN HARDWARE DE DETECTOR DE OJOS SOBRE VIDEO EN TIEMPO REAL"
Title: "Hardware Implementation of a Real Time Eye Detection in Video Signals **"**

Presentado a examen el día: 26 de Septiembre de 2017

para acceder al Título de

INGENIERO DE TELECOMUNICACIÓN

Composición del Tribunal:
Presidente (Apellidos, Nombre): Íñigo Ugarte Olano
Secretario (Apellidos, Nombre): Pablo Pedro Sánchez Espeso
Vocal (Apellidos, Nombre): Víctor Fernández Solórzano

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente Fdo.: El Secretario

Fdo.: El Vocal Fdo.: El Director del PFC
(sólo si es distinto del Secretario)

Vº Bº del Subdirector Proyecto Fin de Carrera Nº
(a asignar por Secretaría)

Agradecimientos

A Pablo por su mirada, su tiempo, su paciencia, su despacho, su mesa, su ordenador… pero sobre todo, por su optimismo.

A Iñigo y a Álvaro por la ayuda y el apoyo.

Mi agradecimiento más especial es para María, ya que este trabajo es tan suyo como mío. Este documento estaría en blanco sin el apoyo, el esfuerzo, el cariño y el ánimo continuos, incondicionales e inagotables de mi compañera.

Índice

Ilustraciones

Líneas de código

1. Introducción

La visión artificial es un ámbito tecnológico al que se está dedicando un gran esfuerzo de desarrollo. Muchas de las aplicaciones punteras que nos son ofrecidas en la vanguardia de las nuevas tecnologías se apoyan en mayor o menor medida en algoritmos y sistemas de procesado de señales de video. Fijémonos por ejemplo en los pilotos automáticos de coches como los que ofrece la empresa Tesla, entre otras, capaz de realizar una conducción eficiente y segura por cualquier carretera sin la intervención de una persona. Las interfaces Hombre-Máquina de los últimos modelos de videoconsolas de juegos que son capaces de interpretar nuestros movimientos a través de una cámara, los sistemas de lectura automática de matrícula en los aparcamientos o la gran cantidad de aplicaciones AOI (automatic optical inspection) o de control de procesos que encontramos hoy día en entornos industriales son algunos más de los muchos ejemplos que podemos contar.

Aunque hay grandes avances al respecto, la visión artificial es un campo que aun ofrece muchas oportunidades y donde día a día aparecen nuevas aplicaciones y mejoras debido al desarrollo de nuevos algoritmos de procesado más eficientes y adaptativos y el uso de plataformas más potentes y capaces de procesar grandes cantidades de información en tiempo real. Y es que la complejidad y la magnitud de datos que es necesario procesar en cada una de estas aplicaciones hacía imposibles hasta hace bien poco, los resultados que hoy están al alcance de la mano.

La detección de objetos es uno de los principales hilos de desarrollo dentro de la visión artificial. El modo en que las personas interpretamos el mundo a través de nuestros ojos está basado principalmente en la capacidad de segmentar y dar entidad propia a cada uno de los objetos físicos que se presentan ante nuestra vista. Sin embargo, es difícil de discernir el mecanismo que nuestro cerebro realiza para que esta capacidad sea tan natural para nosotros.

Tratamos de conseguir que las máquinas adquieran e incluso mejoren nuestras capacidades para poder realizar tareas más rápida y precisas que nosotros. Sin embargo, en lo que a la visión artificial se refiere y especialmente en la detección de objetos, estas capacidades, aun parecen estar lejos. Los algoritmos de detección requieren gran cantidad de recursos aun cuando se aplican a sistemas sencillos y concretos. Esto hace

que en general sean sistemas lentos y poco adaptativos, con grandes dificultades para llevar a cabo tareas en tiempo real.

El trabajo que abordamos explota la implementación eficiente de un algoritmo de detección. Trabajaremos en el desarrollo de un producto, que a través de los nuevos SoC (System on Chip) programables a nivel tanto hardware (FPGA) como software (CPU), permita ejecutar aplicaciones de visión de alto rendimiento. Además, trabajaremos en un algoritmo de detección rápido, ligero y modular, con un alto grado de eficiencia, que se procesará concurrente con la aplicación principal que consuma sus resultados.

Hemos elegido los ojos como objeto a detectar porque pensamos que ofrece un amplio abanico de posibilidades de aplicación. A partir de la detección de ojos podemos diseñar un sistema de interfaz hombre-máquina que reemplace a un ratón o que nos permita pulsar teclas sobre una pantalla sin tocarlas. Los ojos son un buen punto de referencia para localizar, medir o determinar la posición de una persona. Dicen que una persona que mira a los ojos cuando habla es una persona que infunde confianza, por eso hemos decidido que nuestra máquina dirija ahí su mirada.

A lo largo del siguiente trabajo:

- Presentaremos las especificaciones iniciales.
- Haremos un recorrido sobre el estado del arte en el campo de la detección de patrones así como de las herramientas y plataformas de desarrollo.
- Expondremos el proceso de Diseño y las estrategias de Verificación que se han llevado a cabo a lo largo del trabajo.
- Por último, mostraremos los resultados alcanzados.

2. Especificaciones

El objetivo principal de este trabajo es:

Implementar un sistema hardware de detección de los ojos de una persona utilizando un detector de objetos de tipo "Cascade Boosted Features".

En primer lugar será necesario desarrollar una plataforma de desarrollo para poder implementar el algoritmo a nivel de hardware. Basaremos nuestro diseño en un SoC de Xilinx de la familia zynq7000. Esta plataforma deberá ser capaz de recibir una señal de video de resolución y frame-rate variable y albergar el IP core con el detector necesario para procesarla.

Utilizaremos la placa de desarrollo ZedBoard [3] junto con el módulo AES-FMC-HDMI-CAM-G [4] para la interfaz de entrada de video.

El entorno de desarrollo escogido es *"VIVADO HLx Design Suite"* [5] de Xilinx para el diseño, la síntesis, el depurado y la implementación hardware y el "*Xilinx Software Development Kit (XSDK)*" [6] para la programación y el depurado de los drivers y la aplicación.

En segundo lugar diseñaremos, simularemos y sintetizaremos el detector de ojos. Para la descripción y la síntesis utilizaremos "High Level Synthesis" (HLS). Emplearemos el entorno *"VIVADO HLx Design Suite"* [5] también para esta tarea. El resultado será un IP core que podamos utilizar y portar a diferentes diseños y aplicaciones. La interfaz de dicho IP será de tipo AXI4 Stream [7] para la recepción de video y AXI4_Lite para la comunicación con la CPU.

El detector que desarrollaremos es de tipo WaldBoost [14] entrenado con el algoritmo AdaBoost [2][8][9]. El detector constará de una secuencia de 256 clasificadores sencillos de tipo LRD (Local Rank Differences) [10][11], relacionados en cascada, y trabajando sobre muestras de 25 x 25 pixels.

3. Estado del arte

3.1. Algoritmo

Desde la propuesta de *Paul Viola* y *Michael Jones* en 2001 [1] el método “Cascade Boosted Features” es sin duda el más utilizado para la detección de imágenes en señales de video y el primero que permite hacerlo en tiempo real. Este método utiliza un algoritmo de aprendizaje (llamado AdaBoost o alguno de sus derivados), que permite entrenar una serie de clasificadores muy eficientes que pueden ser asociados para obtener un clasificador mucho más robusto. Este algoritmo fue introducido en 1997 por *Yoav Freund* y *Robert E. Schapire* [2]. AdaBoost es un método iterativo, que basándose en una serie de clasificadores iniciales y realizando un estadístico sobre el error acumulado en una serie de muestras controladas, permite seleccionar aquellos clasificadores que ofrecen mejores resultados convergiendo hacia una solución óptima.

Una vez entrenado el detector, el resultado es un conjunto de clasificadores que en principio pueden no ser muy precisos pero que al aplicarlos en cascada ofrecen una eficiencia muy alta. Estos clasificadores se encargan de extraer de la imagen una serie de características y son denominados de Hipótesis débil por su baja precisión. Para que el detector produzca una salida positiva, la salida de cada uno de los clasificadores que lo componen se compara con un umbral (esta es la hipótesis fuerte) que a su vez debe tener un resultado positivo. Cuando todos los clasificadores resuelven positivamente la comparación con cada umbral, se compara el resultado final y, si se supera el umbral, se considera detectado el objeto.

El detector se suele aplicar sobre segmentos de la imagen de un tamaño determinado. El algoritmo trabaja de manera iterativa desplazando los segmentos de imagen a través de la imagen completa pixel a pixel. Como el tamaño del objeto a detectar debe coincidir con el tamaño de la muestra, resulta necesario realizar operaciones de escalado para poder detectar objetos de distinto tamaño o a diferentes distancias. Para realizar el escalado pueden utilizarse dos técnicas principalmente: realizar una pirámide de escalado sobre la imagen original o cambiar el tamaño de la

muestra. En otras ocasiones, se mejora la eficiencia del escalado combinando las técnicas anteriores [11].

Para cada una de las muestras que componen la imagen se extraerá una serie de características. Estas características son obtenidas al evaluar la función de hipótesis para cada uno de los clasificadores que componen el detector (que a su vez son llamados “stages”). Los “stages” estarán ordenados de manera que los primeros de ellos descarten rápidamente la gran mayoría de las muestras que no contienen objetos. De esta manera conseguimos que los requisitos computacionales se mantengan bajos. Cada hipótesis está compuesta por los parámetros de extracción necesarios (posición, tamaño, orientación, rangos, etc) así como el umbral de comparación para poder tomar la decisión (hipótesis fuerte).

Uno de los métodos de extracción de características más utilizado por su sencillez son los filtros de Haar, propuesto por Viola and Jones[1] en su trabajo. Este método consiste en calcular la diferencia de intensidad en pequeñas secciones de la imagen. Se establecen una serie de patrones predeterminados como los de la imagen.

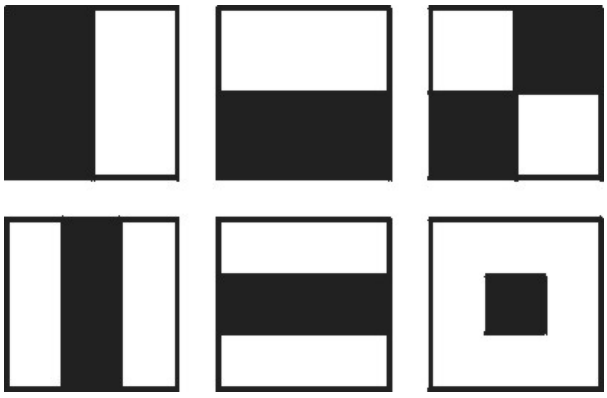


fig 1: Filtros de Haar

El resultado de la función será la diferencia entre la suma de intensidad de los pixels en la sección negra y la sección blanca:

$$f(x) = \sum_{W} I_{px} - \sum_{B} I_{px}$$

Estaremos calculando por lo tanto diferencias abruptas dentro de la imagen. Si aplicamos alguno de los filtros sobre toda una imagen, el resultado será similar al cálculo del gradiente de la imagen en la dirección que marque el filtro.

Otro de los métodos más sencillos es el Local Binary Pattern (LBP)[13]. Las hipótesis LBP extraen características estructurales de la imagen. Cada caracterizador LBP está compuesto por un grid de 3x3 y el resultado es un número de ocho bits en el que cada bit corresponde con el resultado de la comparación de cada pixel de los extremos del grid con el pixel central.

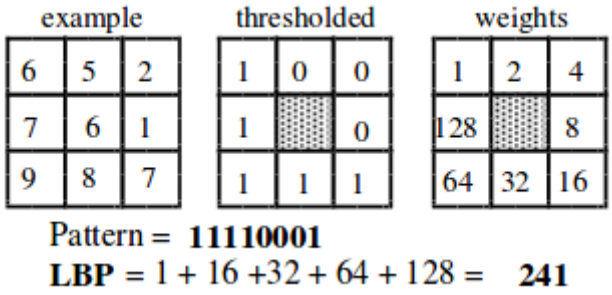


fig 2: Local binary pattern

Dicho número genera un histograma simplificado de los píxeles vecinos. Este procedimiento tiene la ventaja de que es invariante con la iluminación y el contraste, por lo que no es necesario normalizar la imagen. Además, resulta más eficiente que los filtros de Haar. Un caracterizador LBP se caracteriza por la posición dentro de la muestra del feature y por el tamaño en pixels de las celdas que componen el grid.

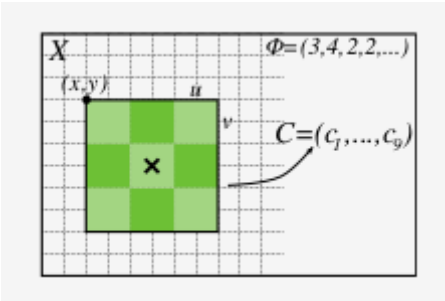


fig 3: caracterización de un stage LBP

Otro clasificador basado en la geometría es el Local Rank Differences (LRD)[13]. En este caso también utilizaremos un grid de 3x3 celdas. El LRD, además de la posición en la imagen y el tamaño de cada celda del grid, se caracteriza por dos posiciones dentro del grid (A y B). Compararemos todos los pixels con cada uno de estos dos pixeles de

referencia. El resultado será la diferencia entre el número de comparaciones positivas con un valor y con el otro. El resultado por lo tanto estará en un rango de -8 a 8.

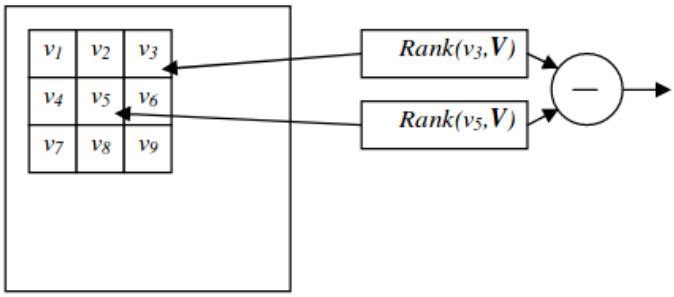


fig 4: diagrama de un algoritmo LRD

En este caso, aunque el algoritmo es algo más complicado, es también invariante con el contraste y la exposición y además, el número de “stages” necesarios para conformar un detector es mucho menor que con los filtros de Haar, por ejemplo.

Una variante del LRD, el Local Rank Pattern (LRP), obtiene aún mejores prestaciones según Hradis, Herout y Zemcik [10]

Una vez extraidas las características con los clasificadores (features) debemos evaluarlas y tomar las decisiones correspondientes. Es por tanto que cada “stage” debe ir acompañado por una función de toma de decisión. En el caso de los filtros de Haar, la función más común es la comparación con un valor umbral. Cada “stage” devuelve el valor de diferencia de intensidad en un área y una posición específica de la imagen como ya hemos dicho. Este valor se comparará con un umbral ajustado a la probabilidad estadística de que una imagen que represente al objeto buscado produzca ese resultado en ese lugar de la imagen. Es decir, si todas la muestras positivas que hemos utilizado para el entrenamiento devolverían una diferencia de intensidad mayor que θ en la posición [x, y] con un filtro de dirección δ y tamaño [u, v], determinaremos que el resultado es positivo si la salida del clasificador H(δ, x, y, u, v) es mayor que θ.

Si nuestro detector se basa en clasificadores de tipo LBP la función de toma de decisión será probablemente la comparación del histograma de salida con una máscara. En el caso LRD, la función de toma de decisión comparará con un rango.

Para que el detector sea realmente efectivo, no podemos utilizar “stages” con características aleatorias. Es en este aspecto en el que el algoritmo de aprendizaje y entrenamiento del detector es fundamental. El algoritmo de machine-learning AdaBoost nos permite seleccionar aquellas features con las características más relevantes, que son

aquellas donde existe mayor diferencia entre las muestras positivas y negativas. Además, se ordenarán de forma que en las primeras iteraciones se descarte el mayor número de muestras en las cuales no se encuentra el objeto a detectar. El algoritmo 'AdaBoost' realizará un estudio estadístico aplicando los potenciales clasificadores a una gran base de datos con imágenes con y sin el objeto a detectar y ofrecerá una función de toma de decisión optimizado, para cada uno de los "stages" que formarán parte del detector [2].

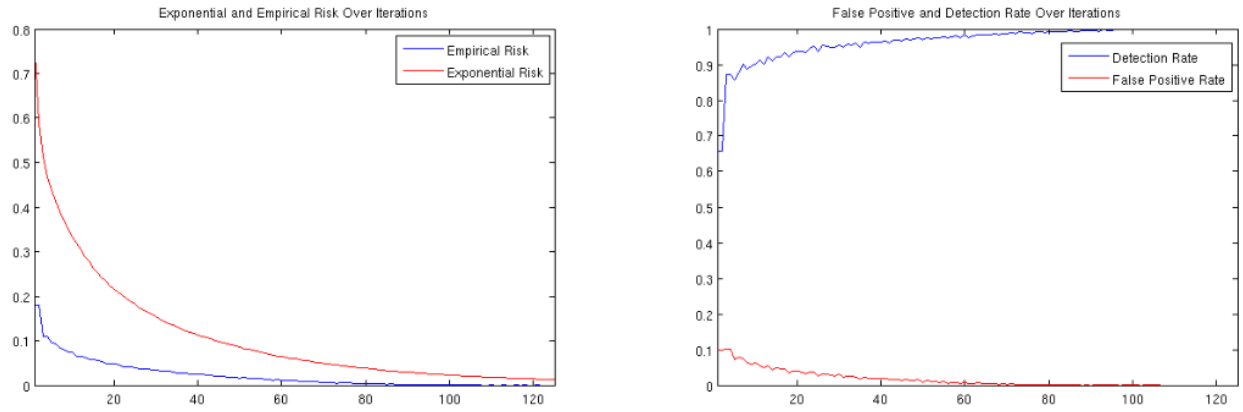


fig 5: Análisis de los indices de error con respecto a las iteraciones de un proceso de entrenamiento AdaBoost de un detector facial utilizando 2500 imágenes positivas y 2500 negativas. Extraído de [16]

Cada "stage" por separado resulta ser un detector con una resolución relativamente baja, por lo que se producirían muchos falsos positivos. Sin embargo, cuando encadenamos centenas de estos clasificadores, es decir, cuando extraemos centenas de características relevantes del objeto, la probabilidad de error se reduce drásticamente y conseguimos detectores con un altísimo nivel de eficacia.

Entendemos entonces que este tipo de algoritmos suponen una ventaja notable en cuanto a eficacia. Pero además de esto, el algoritmo es extremadamente eficiente en cuanto a cómputo. Aunque a-priori parezca un algoritmo denso y pesado, con muchas operaciones, en realidad tenemos que entender que cada una de estas operaciones son operaciones extremadamente sencillas, que se reducen a sumas y comparaciones en la mayoría de los casos. Estas operaciones son muy fáciles y rápidas de computar. Además, se realizan sobre superficies de la imagen relativamente pequeñas. Cada "stage" se aplica en la gran mayoría de los casos sobre una superficie relativamente pequeña de la muestra y por lo tanto de la imágen. Esto permite hacer los cálculos de una manera muy localizada y reducir el número de operaciones de store/load.

Además de esto, la naturaleza del algoritmo, que permite descartar en las primeras iteraciones la mayor parte de las muestras sin información relevante, hace que el número

de “stages” procesados sea en realidad relativamente bajo para la mayor parte de las imágenes e incluso muy bajo para imágenes que no contienen el objeto a detectar.

El resultado es un algoritmo que requiere un esfuerzo de cómputo considerablemente menor que algoritmos de detección tradicionales basados en segmentación y/o correlación que normalmente requieren operaciones complejas y trabajan sobre secciones de imagen grandes.

Sin embargo, este tipo de detectores tiene algunos puntos débiles como son:

- Trabajan solo con imágenes frontales: las características de la imagen solo son comunes para objetos vistos en una posición (en la que fue entrenado el detector) Las imágenes de ese objeto visto lateralmente o desde atrás presentarán características diferentes y el objeto no será detectado.
- La rotación del objeto dentro de la imagen es crítica: si la imagen se presenta girada el detector no funciona. Existen algunas propuestas para mejorar este aspecto, pero en general, este tipo de procedimientos es muy sensible a la posición de los clasificadores.

Partiendo del algoritmo “Cascade boosts features” de Viola & Jones, Sochman y Matas proponen en 2005 [14] un algoritmo que mejora tanto el indice como el tiempo de detección. Este algoritmo, llamado Waldboost, utiliza AdaBoost para seleccionar y ordenar los features. Después añade un procedimiento llamado SPRT (sequential probability ratio test) que utiliza la función densidad de probabilidad de cada “stage” para realizar la toma de decisión.

Sochman & Matas demuestran que, encadenando decisiones basadas en la densidad de probabilidad en vez de en valores absolutos de probabilidad el error se reduce con las iteraciones. La función de toma de decisión para cada “stage” está basado en este caso en dicha función de densidad de probabilidad enlazada. Por lo tanto el resultado no depende únicamente del “stage” actual sino también de los anteriores.

Este algoritmo requiere que las funciones de densidad de probabilidad de cada clasificador sean conocidas a-priori. Cada hipotesis por lo tanto depende, además de la posición, el tamaño, el tipo de operador y el valor umbral de decisión, de la función densidad de probabilidad: $H(\delta, x, y, u, v, \alpha_{(x)})$.

En este caso, al depender cada una de las iteraciones de las anteriores, la secuencia de los “stages” se convierte en algo crítico.

Method	WB	VJ[8]	Li[3]	Xiao[11]	Wu[10]
#wc	400	4297	2546	700	756
$\bar{T}_S$	10.84	8	(18.9)	18.1	N/A

fig 6: Resultados reportados por Sochman & Matas comparando su algoritmo con otros dentro del estado-del—arte[14].

En la tabla de la *fig 6* podemos ver como la relación entre el número de stages y el tiempo de cómputo del detector WaldBoost es mucho mejor que para otros detectores. Aunque el sistema de Viola and Jones es ligeramente más rápido, el detector WaldBoost es más de diez veces más compacto.

La aplicación del algoritmo WaldBoost para la detección sigue el siguiente esquema:

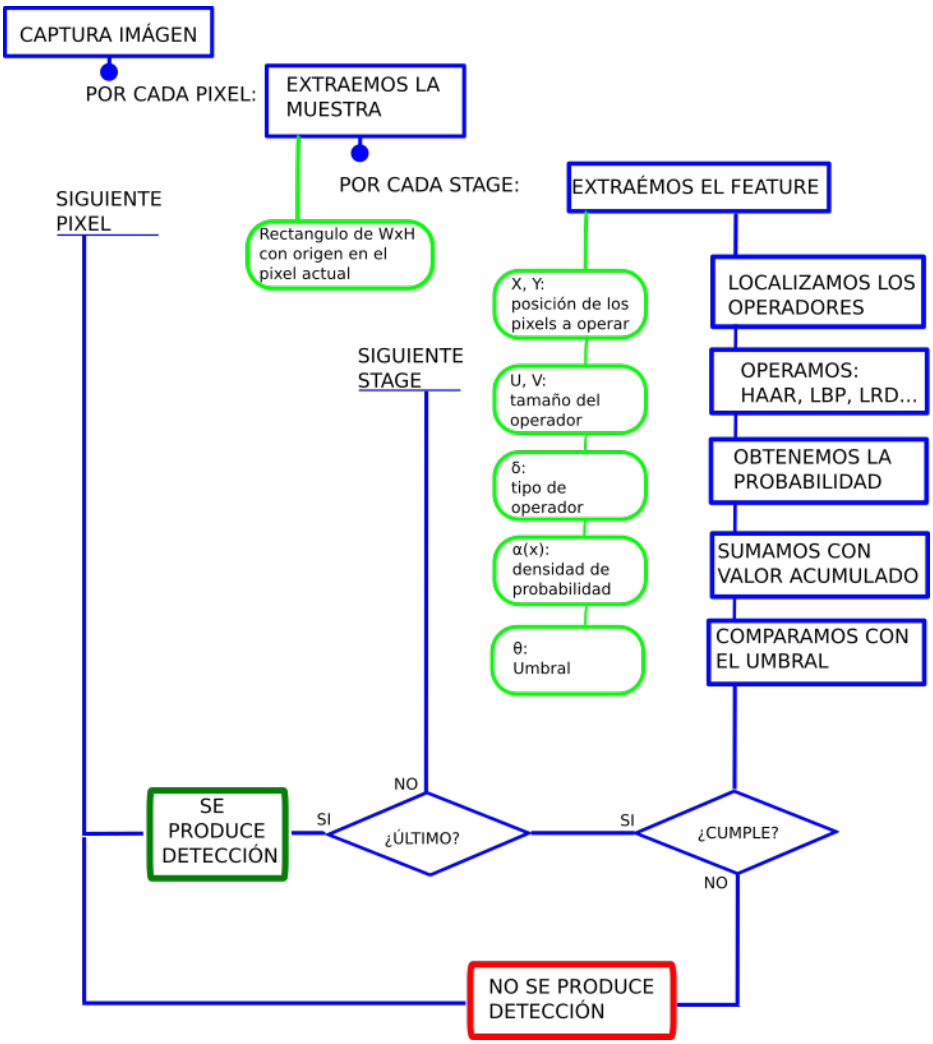


fig 7: Algoritmo WaldBoost de detección de objetos

3.2. Estudio de las implementaciones realizadas hasta ahora

A la hora de implementar estos detectores en sistemas reales nos encontramos, con:

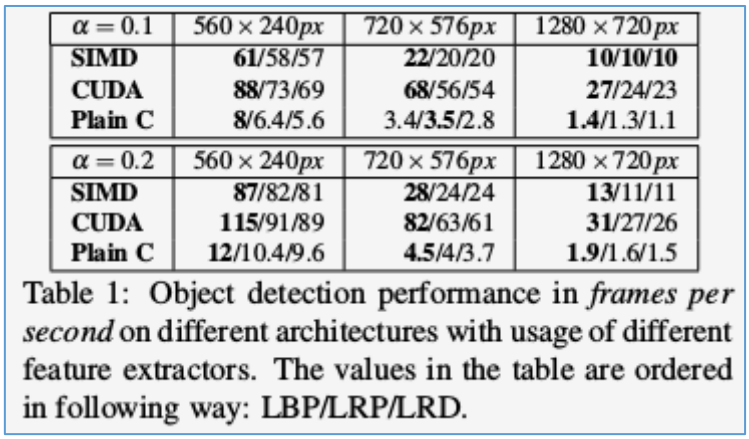
1. Operaciones muy sencillas: que ofrecen una respuesta rápida y no requieren muchos recursos de computación
2. Operaciones localizadas: que reducen la necesidad de realizar muchos accesos a memoria.

Esto permitirá desarrollar sistemas con altos valores de “frame-rate” en el procesado de imágenes pudiendo obtener sistemas de tiempo real sin grandes requisitos computacionales. Además, no será necesario almacenar “frames” completos en memoria lo que hace posible procesar directamente sobre una secuencia de imágenes (stream) de entrada.

Hay varios esfuerzos por implementar detectores de este tipo en hardware, ya sea en FPGAs o sistemas ASIC. La mayoría de ellos utiliza filtros Haar en “imagen integral”, un sistema de pre-procesado que permite realizar los cálculos con un tiempo de respuesta constante. En el caso del trabajo llevado a cabo por Zemčík, Juránek, Musil, Musil y Hradiš vemos como utilizando WaldBoost podemos obtener grandes resultados sobre una Spartan 6 [13].

También se han propuesto implementaciones que mejoran los resultados de la detección en aplicaciones ejecutadas a nivel de software paralelizando parte de las operaciones mediante GPU[19] (por ejemplo programando el código con CUDA). En este caso utilizaríamos las instrucciones específicas de las tarjetas gráficas para resolver una parte importante del algoritmo sin la intervención directa de la CPU.

El uso de instrucciones Multimedia, como MMX, que incluyen muchas de las CPUs actuales así como el juego de instrucciones SSE de Intel también permiten mejorar el rendimiento de los detectores permitiendo realizar operaciones en paralelo sobre los datos. Juránek, Zemčík y Herout en su trabajo “*Implementing the Local Binary Patterns with SIMD Instructions of CPU*” [12] nos ofrecen la gráfica de la figura 8 en la que puede verse la diferencia de ejecutar un detector sobre tres implementaciones distintas: utilizando instrucciones SSE, utilizando CUDA y con la ayuda exclusiva de la CPU. Los autores han realizado el experimento utilizando operadores de tipo LBP, LRD y LRP. Podemos ver como la paralelización del procedimiento permite conseguir frame-rates hasta diez veces mayores para el mismo programa.



$\alpha = 0.1$	$560 \times 240px$	$720 \times 576px$	$1280 \times 720px$
SIMD	**61**/58/57	**22**/20/20	**10/10/10**
CUDA	**88**/73/69	**68**/56/54	**27**/24/23
Plain C	**8**/6.4/5.6	3.4/**3.5**/2.8	**1.4**/1.3/1.1
$\alpha = 0.2$	$560 \times 240px$	$720 \times 576px$	$1280 \times 720px$
SIMD	**87**/82/81	**28**/24/24	**13**/11/11
CUDA	**115**/91/89	**82**/63/61	**31**/27/26
Plain C	**12**/10.4/9.6	**4.5**/4/3.7	**1.9**/1.6/1.5

Table 1: Object detection performance in *frames per second* on different architectures with usage of different feature extractors. The values in the table are ordered in following way: LBP/LRP/LRD.

fig 8: Comparativa de diferentes implementaciones de un detector facial extraído del trabajo "Implementing the Local Binary Patterns with SIMD Instructions of CPU"[12]

3.3. Desarrollo de hardware programable

En el momento de redacción de este proyecto, el desarrollo hardware y la electrónica en general se encuentra en un punto de inflexión importante. A partir de 2005 con la aparición de las primeras placas arduino, se generaliza el uso de hardware libre en el campo del desarrollo electrónico. Por ello se ha democratizado el acceso a sistemas de desarrollo que antes tenían un coste muy elevado y que requerían de conocimientos específicos. Como consecuencia, las curvas de aprendizaje se reducen considerablemente y se facilita el desarrollo de sistemas a niveles más altos de abstracción. En 2012 con la llegada de SBC (Single Board Computer) de bajo coste como la Raspberry pi, aparece la posibilidad de embeber sistemas completos con una alta complejidad tanto hardware como software por menos de 40€. La llegada de las familias Spartan y Ciclone de FPGAs, con precios considerablemente más reducidos que sus predecesoras, permitió la comercialización de placas de desarrollo por valor de unas pocas centenas de euros en lugar de millares. Actualmente podemos encontrar placas de desarrollo para estas familias en torno a 20€ en internet.

En el campo de las FPGA, el cambio más significativo fue sin duda, la llegada en 2009 de la familia Zynq de Xilinx que integraba en el mismo chip un core multi-CPU con arquitectura ARM, memoria e interfaces de entrada salida (Ethernet, USB, I2C, CAN, UART, GPIO, etc) junto con un campo de celdas programable completo (FPGA). La aparición en 2013 de las plataformas de desarrollo integradas que lo acompañan (Vivado, HLS Vidvado, SDSoC, etc) permiten trabajar en el desarrollo integral de sistemas en unos

niveles de abstracción increíblemente altos. Esto, junto con la constante oferta de nuevas placas de prototipado y desarrollo a precios cada vez más reducidos (microzed por 199€, Zynqberry TE0726-03M por 109€) permite llevar a cabo proyectos de casi cualquier envergadura y con bajo coste inicial.

3.4. FPGA y SoC de Xilinx

La empresa Xilinx inc. fue la primera empresa en lanzar una FPGA en 1984 y actualmente es líder del sector de la lógica programable con más del 50% de la tasa de mercado.

En el momento de la presentación de este trabajo, la oferta más avanzada de FPGA de Xilinx se enmarca dentro de su “Serie 7” donde se incluye a las familias Atix, Kintex, Virtex y Spartan. En la siguiente tabla podemos ver alguna de sus características principales.

	Logic Cells	FF	BRAM(Kb)	I/O	DSP
SPARTAN$_7$	6k-102k	7k-128k	180-4320	100-400	10-160
ARTIX$_7$	12k-215k	16k-269k	720-13140	150-500	40-740
KINTEX$_7$	65k-477k	82k-100k	4860-34380	300-500	240-1920
VIRTEX$_7$	326k-2000k	408k-2443k	27000-67686	300-1200	1120-3600

La serie UltraScale ofrece FPGAs de las familias Kintex y Virtex con tecnología 20nm y 16nm y unas prestaciones aún mayores.

La familia Zynq-7000 integra en el mismo chip una FPGA de tipo Artix o Kintex con al menos un procesador ARM Cortex-A9. Esto permite aunar en un solo chip CPU y PL (Programable Logic).

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

fig 9: Segmento del "Zynq7000 Product Selection Guide"

La familia Zynq-UltraScale se caracteriza por un mayor indice de integración. En estos chips conviven GPUs con CPU de tipo ARM Cortes-A54 (core de altas prestaciones y 64 bits) con ARM Cortex-R5 (de bajo consumo y comportamiento predecible para aplicaciones críticas en tiempo real).

3.5. Placas de desarrollo para Zynq7000

Xilinx ofrece dos placas de desarrollo para su familia zynq-7000

1. La **ZC702** está basada en el chip XC7Z020.

   La placa dispone entre otras cosas de:

   - 1GB de memoria DDR3 RAM
   - Interfaces UART, USB OTG, I2C y CAN
   - Giga Ethernet Phy.
   - Expansión a través de dos conectores FMC y tres PMOD
   - Conector y driver de salida HDMI
   - Programador (JTAG) integrado.
   - Interfaz SD-CARD

2. La **ZC706** está basada en el chip XC7Z0445

La placa añade a las características de la ZC702, entre otras cosas:

- Interfaces SODIM y PCIe
- conectores SMA y SFP+

Estas placas tienen un precio según la página oficial de Xilinx de 895$ y 2495$ respectivamente

Digilent y Avnet ofrecen una alternativa mas económica a las placas de Xilinx, entre la que destaca la **ZedBoard,** basada en el chip XC7Z020, al igual que la ZC702.

La placa cuenta con prácticamente las mismas prestaciones que la ZC702 aunque solo dispone de 512MB de memoria DDR RAM y una interface FMC

Por otro lado, la ZedBoard dispone de más conectores PMOD e incluye una interface y CODEC I2S de audio, una interface VGA y un pequeño display OLED.

Zedboard tiene un precio aproximado de 475$. Avnet distribuye también versiones más compactas basadas en diferentes chips zynq: minized, nanozed, microzed y ultrazed con diferentes factores de forma, conectores e interfaces para adaptarse a desarrollos más concretos.

Trenz electronic también distribuye varias versiones de placas de desarrollo para diferentes cores zynq-7000. Su familia TE07XX ofrece multitud de opciones en cuanto a factor de forma, interfaces y prestaciones. Destacamos aquí el modelo **TE0726** que ofrece un factor de forma, interfaces y distribución compatibles con el SBC Raspberry pi. La TE0726 utiliza un XC7Z010 que incluye:

- 512MB de memoria DDR3 RAM
- 100M Ethernet phy
- 4 USB
- programador (JTAG) integrado
- UART and 26MPIO conector
- HDMI output
- SD-CARD interface
- DSI y CSI-2 conectors (para display y cámara respectivamente)

Esta placa se oferta por 109$. Por otro lado las placas **Zybo** y **ArtyZ,** basadas en XC7Z010 y XC7Z020 respectivamente, son una nueva opción de Digilent por menos de 200$. Ambas disponen de:

- 512MB de memoria DDR3 RAM
- Giga Ethernet phy
- USB OTG, UART, I2C, CAN
- programador (JTAG) integrado
- HDMI output/input
- SD-CARD interface

Estas placas no cuentan con interfaz FMC aunque disponen de otros conectores como Pmod o “Arduino shield compatible”.

Tanto Avnet, como Digilent y Trenz ofrecen también un alto número de módulos de expansión con interfaces FMC, Pmod, etc para extender las funcionalidades y añadir hardware adicional a sus placas de desarrollo. Algunos de estos módulos permiten añadir a nuestros sistemas interfaces de video, LVDS, displays, transceptores RF, drivers de motores, expansores de I/O, etc.

3.6. Entornos de desarrollo

El proceso de desarrollo basado en FPGA es complejo. Si el desarrollo incluye además sistemas microprocesados y software, la complejidad se multiplica.

La gran mayoría de los proyectos de desarrollo actuales basados en FPGAs y/o SoC está dividido en tres procesos diferenciados:

1. **Implementación de las funciones específicas**: El diseño precisa módulos o funciones básicas que resuelva las tareas que el sistema final requiere. Estas funciones suelen individualizarse y desarrollarse como bloques autónomos, modulares y escalables. A estos bloques se les denomina *bloques de propiedad intelectual (IP) o cores.* Estos IP cores se desarrollan utilizando, bien un lenguaje de descripción de hardware (HDL) como puede ser VHDL o VERILOG, bien lenguajes de más alto nivel como C, C++, Phyton, SystemC, etc. Estos últimos precisan de una síntesis HLS (High Level Synthesis) [20]. HLS utiliza lenguajes de programación secuenciales normalmente empleados en desarrollo software. Una herramienta de síntesis ‘traduce’ el programa de alto nivel a descripción HDL para finalmente realizar una síntesis a nivel RTL.

El resultado de todo este proceso suele ser encapsulado para su exportación y reutilización de un modo muy similar a como se encapsula parte del código de las aplicaciones software utilizando librerías. De este modo, el resultado final se denomina Intellectual Property Core (IP Core).

2. **Desarrollo de la plataforma (shell creation)**: una vez definidos todos los bloques elementales que formarán parte del diseño final, se ha de construir la estructura final (shell) del hardware. Para ello podemos utilizar, además de los elementos desarrollados en la etapa ateriormente descrita, otros bloques o IP Cores disponibles en el repositorio de la herramienta o de terceros proveedores. Muchos de estos IP Cores describen funcionalidades generales como son la interfaz con el PS (programable sistem) o CPU, interfaces específicas de determinadas Boards, DMAs, FIFOS, generadores de reloj y de reset, etc. También proporcionan funciones concretas como conversiones de formato de video, multiplicadores, pilas TCP/IP, etc.

La relación e interconexión de estos bloques se describe de nuevo a través de lenguajes de tipo HDL y de la descripción de *constrains* y reglas específicas de cada herramienta. Algunas herramientas disponen de asistentes y herramientas gráficas para poder llevar a cabo esta tarea de una manera más amigable y sencilla para el usuario ya que es en verdad un trabajo arduo. La aplicación IP Integrator de la plataforma VIVADO de Xilinx es un buen ejemplo de asistente gráfico para la definición de sistemas a través de IP Cores en la que la definición final del sistema se realizaría como si cableáramos un esquemático con diferentes componentes electrónicos.

El resultado de este proceso será la implementación del sistema completo a nivel de transferencia de registros (RTL) y la elaboración del fichero binario con dicha implementación para poder programar la FPGA. Este fichero final se denomina Bitstream.

3. **Desarrollo del Software**: La gran mayoría de los sistemas electrónicos modernos dependen en mayor o menor medida del uso de programas software que se ejecutan en una o varias CPUs. Esto implica la necesidad de desarrollar la aplicación que vaya a residir en la memoria de programa. La tercera de las capas de desarrollo consiste precisamente en este proceso y se llevará a cabo utilizando algún lenguaje de programación como ensamblador, C, C++, java o Phyton.

A lo largo de este procedimiento se obtendrá el software necesario en todos los niveles: desde el sistema operativo, pasando por los drivers de cada uno de los elementos que compondrán la plataforma hardware desarrollada en la fase anterior, hasta la aplicación final del usuario. El resultado será uno o varios ficheros binarios que serán cargados en memoria.

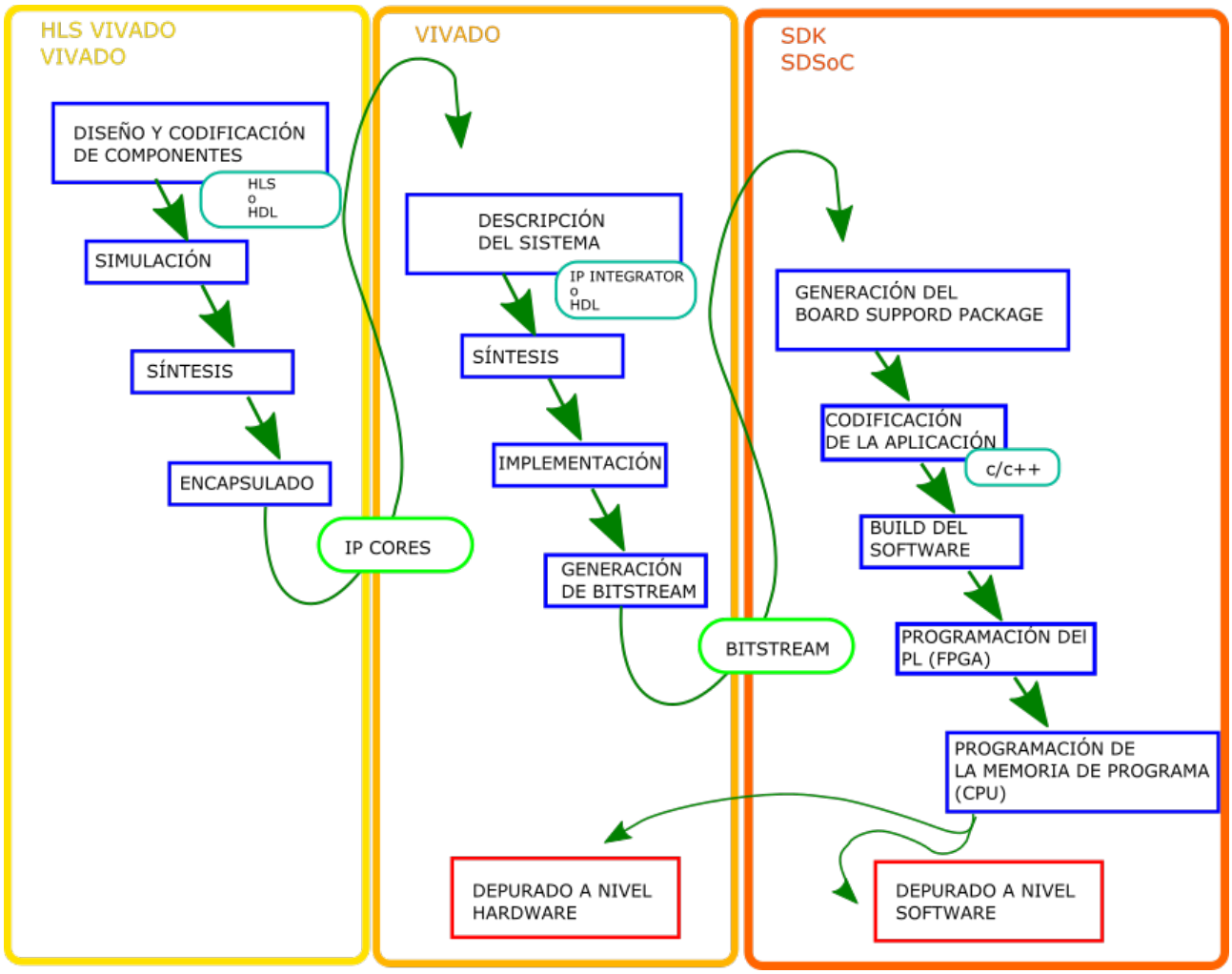


fig 10: Desarrollo de sistemas FPGA/SoC

El **paradigma** de desarrollo de sistemas **ha cambiado** considerablemente en los últimos años. La demanda de sistemas cada vez más complejos y polivalentes desborda constantemente los límites de densidad, rendimiento y eficiencia energética. La construcción de los nuevos sistemas electrónicos es un verdadero reto para los equipos de desarrollo que han de alcanzar los objetivos de producción con plazos cada vez más estrechos debido a la vertiginosa velocidad del mercado [22].

La metodología tradicional de implementación hardware requería segmentar el diseño final en pequeños bloques funcionales para poder describirlos a través de lenguajes cercanos a RTL (Register Transfer Level) para después sintetizarlos e implementarlos de forma recursiva hasta cumplir los requisitos del sistema. Este proceso se repetiría una y otra vez para cada una de las secciones definidas previamente. Esta metodología requería de recursos humanos muy especializados con un altísimo “Know

How” tecnológico. Para poder acelerar el desarrollo era necesario contar con un equipo cuantioso incrementándose considerablemente los costes de diseño. La gran mayoría del tiempo se utilizaría en la generación de las líneas de código y su implementación en detrimento de la verificación y la optimización.

La metodología de diseño a partir de descripción de alto nivel se basa en buenas prácticas de desarrollo como por ejemplo: la estructuración y modulación del sistema, el diseño basado en Pipelines y el uso de buses y protocolos para interconectar módulos e IPs, entre otras.

Cualquiera de estos métodos era posible y habitual con herramientas como el antiguo ISE de Xilinx y la metodología tradicional, pero las nuevas herramientas potencian y facilitan su uso.

La verdadera ventaja de estas técnicas está en que, actualmente, estas buenas prácticas son inherentes al sistema de desarrollo. Las aplicaciones de desarrollo actuales facilitan adoptar estas buenas prácticas por el modo en que se lleva a cabo el flujo de desarrollo y como trabajan las propias herramientas.

Por otro lado, el paradigma cambia aún más, teniendo en cuenta las posibilidades que las nuevas plataformas brindan. Y es que los entornos se encargan de automatizar muchas de las tareas que si bien son esencialmente tipográficas y extremadamente mecánicas, requerían hasta ahora de un alto porcentaje del tiempo de desarrollo y eran origen de muchos errores en absoluto relacionados con la labor de diseño.

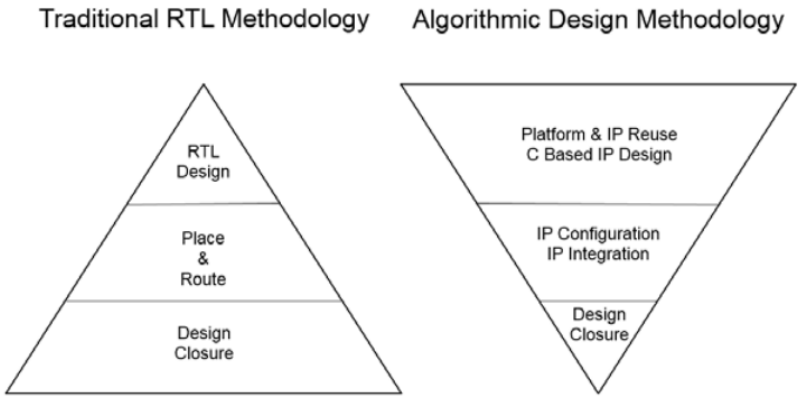


fig 11: Comparación de metodologías de desarrollo

En la gráfica anterior, el área de cada etapa de desarrollo representa el esfuerzo necesario para llevarla a cabo. Podemos ver como en la metodología tradicional el esfuerzo mayor atañía a tareas de implementación mientras que en las metodologías de alto nivel, el esfuerzo se centra en el diseño general del sistema.

La aparición de herramientas de síntesis de tipo High Level Syntesis (HLS) es una de las piedras angulares de esta nueva metodología. Utilizando HLS podemos utilizar lenguajes (C principalmente) para describir funcionalmente y a alto nivel, los bloques funcionales del diseño. La potencia de este tipo de sintesis dentro del conjunto del entorno es impresionante.

- El tiempo de codificación es considerablemente menor
- No es necesario definir los puertos e interfaces a nivel RTL sino que la herramientas de síntesis implementa la interfaz de bus o protocolo. La conexión entre diferentes bloques es prácticamente transparente para el desarrollador.
- Las herramientas HLS permiten realizar simulación a nivel funcional en el propio lenguaje C. Una simulación de un bloque de densidad media a nivel RTL puede tardar varias horas, mientras que utilizando C podemos verificar su funcionalidad en cuestión de segundos.
- En la síntesis HLS, el reloj es también transparente para el usuario. La herramienta de síntesis se encarga de realizar una implementación a nivel RTL que cumpla con la restricción de frecuencia de reloj. De esta forma se eliminan una gran parte de los errores producidos en la síntesis tradicional debido a la aparición de glitchs, skews, lacks, delays, etc.
- El uso de C permite el uso de librerías de alto nivel. Con lenguajes próximos a RTL los componentes de librería son estructuras hardware predefinidas que hay que encajar o conexionar dentro de nuestro código. Sin embargo, una función C no establece una estructura previamente definida y puede sintetizarse de manera diferente dependiendo del código que la acompañe o de la tecnología de la FPGA que la implemente. Además, una llamada a función en C se codifica en pocos segundos sin tener que preocuparte de conexiones, niveles lógicos ni tiempos de retraso. El sintetizador se encarga de todo.
- Una misma codificación en C puede producir diferentes diseños RTL optimizados de diferente manera a través de directivas. Una vez definido funcionalmente nuestro diseño podemos priorizar una baja densidad de recursos a nivel RTL, la velocidad de ejecución, latencia e incluso el uso específico de recursos mediante el uso de directivas de síntesis.

Para el área de la visión artificial, Xilinx incluye en sus herramientas de síntesis HLS la posibilidad de utilizar una versión adaptada de las librerías OpenCV. De esta manera podemos hacer complejos desarrollos hardware utilizando algoritmos sencillos y optimizados de tratamiento de video e imagen con la misma metodología y estructura que lo haríamos para un desarrollo software.

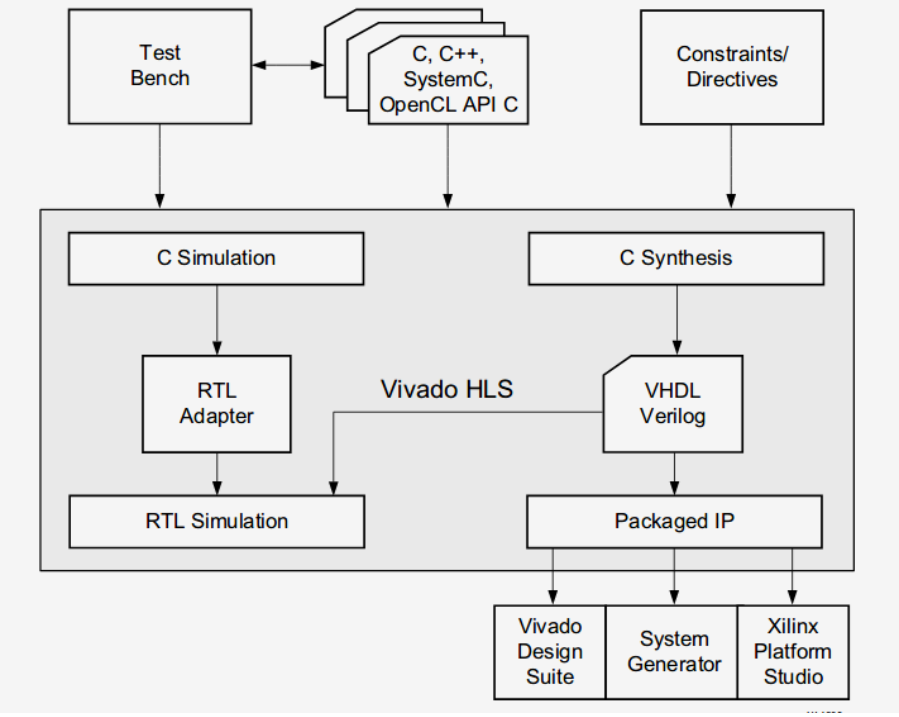


fig 12: Flujo de desarrollo HLS en Vivado.

Xilinx ofrece varias herramientas para poder abordar las fases de desarrollo anteriormente descritas. Aunque todas estas aplicaciones trabajan de manera integrada, conjunta y armonizan a la perfección, no podemos hablar de una plataforma única.

ISE Design Suite es la plataforma tradicional de Xilinx para la descripción, síntesis RTL/lógica e implementación hardware de sus dispositivos programables. Esta completa plataforma permite abordar tanto el desarrollo de componentes como de plataformas hardware utilizando varios lenguajes de descripción HDL.

El sistema ISE dispone de un eficaz simulador RTL que nos permite verificar la funcionalidad del diseño a-priori con mucha resolución. Utilizando el mismo entorno, podemos verificar el funcionamiento a través de diferentes técnicas de depurado hardware.

Sin embargo, desde la comercialización de la Serie 7 de dispositivos programables, Xilinx desaconseja ya el uso de esta plataforma ya que no incorpora las herramientas de última generación que si incluyen otras plataformas más potentes.

VIVADO Design Suite es la plataforma integral que reemplaza a ISE. Esta plataforma incluye toda la funcionalidad de su predecesora pero multiplica su potencia ya que añade entre otras cosas:

- VIVADO HLS: para la síntesis de componentes hardware a partir de lenguajes de programación software
- VIVADO IP Integrator: para la definición de sistemas completos mediante integración de IPs
- DSP system generator: para desarrollo de sistemas de procesado de señal a partir de modelos de MATLAB.

La Suite Vivado se ciñe a la filosofía de desarrollo explicada anteriormente y, aunque permite trabajar a bajo nivel, está pensada para acelerar al máximo el tiempo de desarrollo permitiendo un nivel de abstracción tan alto que acerca el diseño hardware a personas en principio no expertos en este área como pueden ser los programadores de aplicaciones software.

Estas plataformas se integran con otras pensadas para el desarrollo del software sobre las arquitecturas ARM o MicroBlaze.

SDSoC es una plataforma de desarrollo que permite incorporar aceleradores hardware en el desarrollo de software embebido.

En SDSoc podemos indicar la parte del código que queremos que se paralelice utilizando hardware y la herramienta se encarga de crear las interfaces hardware/software apropiadas, sintetizar el código a nivel RTL, implementar el core sobre el bitstream existente, codificar los drivers necesarios y compilar el conjunto haciendo las System Calls pertinentes de manera automática y completamente transparente para el usuario.

Esta característica se denomina Aceleración Automática de Software en Lógica Programable.

SDSoC permite compilar las aplicaciones como BareMetal o junto con sistemas operativos Linux o FreeRTOS lo que permite un rapidísimo desarrollo de sistemas embebidos propios.

Junto con una interfaz JTAG y emulación mediante QEMU en el sistema SDSoC realizaremos todas las funciones de programación, verificación y depurado de un sistema hardware/software.

Un IDE muy similar es también ofrecido por Xilinx y se llama **SDK**. SDK es un entorno de desarrollo software también basado en Eclipse pero no dispone de la potencia ni las prestación de SDSoC para integrar aceleradores hardware.

4. Diseño

4.1. Punto de partida

Comenzamos el diseño del proyecto estableciendo las siguientes premisas:

1. Implementaremos la función de detección partiendo de un detector cuyas características están calculadas a-priori. Dichas características están encapsuladas en un fichero XML que se utiliza para almacenar los parámetros de configuración.

Justificación:

El objetivo del proyecto es poder implementar un detector de manera modular sobre una FPGA. Utilizaremos para el prototipo un detector entrenado previamente y que ha sido probado y testado suficientemente con anterioridad. El detector seleccionado ha sido utilizado ya dentro del departamento en otros proyectos y proporciona un alto grado de confianza.

Tal y como se indica en las especificaciones, es un detector de tipo WaldBoost que trabaja sobre secciones de 25x25 pixels y tiene una profundidad de 256 stages.

2. Utilizaremos para el desarrollo, la plataforma compuesta por ZedBoard + AVNET AES-FMC-HDMI-CAM-G

Justificación:

La plataforma ZedBoard cuenta con los requisitos necesarios para afrontar el proyecto:

- Utiliza como núcleo un SoC de Xilinx [XZ7Z020] que permite un desarrollo integral hardware y software
- Dispone de suficiente memoria DDR RAM: 512MB
- Es rápida y compacta, con un consumo moderado de energía.
- Está perfectamente soportada por todas las herramientas de desarrollo Xilinx
- Dispone de una documentación y soporte adecuados

El módulo AES-FMC-HDMI-CAM-G

- Dispone de interfaz HDMI.
- Puede incorporarse una cámara de video opcional
- Cuenta con documentación y ejemplos que aceleran la curva de aprendizaje
- Se adapta y alimenta perfectamente con la ZedBoard a través de su interfaz FMC

Tanto la ZedBoard como el módulo AES-FMC-HDMI-CAM-G están disponibles en el departamento

3. Utilizaremos para el desarrollo VIVADO Design Suite. El detector será desarrollado en lenguaje C/C++, simulado y sintetizado con VIVADO HLS. Para el desarrollo software utilizaremos el SDK de Xilinx.

Justificación:

- El conjunto formado por VIVADO HLS, VIVADO Design Suite y SDK permite acelerar el tiempo de desarrollo utilizando una metodología de diseño basada en la metodología de diseño UltraFast de Xilinx.
- El detector utilizado ha sido implementado en el grupo de investigación utilizando C/C++ con éxito en plataformas basadas en múltiples procesadores.

4.2. Planificación

La fase de diseño se planifica a-priori con una serie de etapas que se describen a continuación:

4.2.1. Preparación y encapsulado del detector

Como hemos mencionado, el detector es definido por la descripción de sus parámetros, incluidos en una estructura externa. El objetivo del diseño es conseguir un detector codificado en C/C++ y sintetizado a nivel RTL para su implementación en FPGA. Determinamos que los parámetros del detector serán implementados en una ROM integrada dentro del sistema. La forma de definir una ROM en C/C++ para HLS es

declarar una variable constante que no se modifique en tiempo de ejecución. Por ello se seguirá el siguiente procedimiento:

- Definir una estructura en C que contenga todos los parámetros que definen el detector.
- Definir un tipo base de esa estructura
- Crear una variable array del tipo definido con anterioridad con tantos elementos como “stages” compongan el detector WaldBoost. En este caso el número de elementos será 256.
- Inicializar la variable con los valores predefinidos extraídos del fichero XML.
- Crear un fichero de cabecera (.h) que contenga la definición de la estructura, del tipo, la declaración de la variable y la inicialización de sus valores.

La definición de la estructura y del tipo se decide sea de la siguiente manera:

```
typedef struct
{
    unsigned x, y;
    unsigned w, h;
    unsigned A, B;
    short theta_b;
    short alpha[17];
```

código 1: Estructura que contiene los parámetros propios de cada stage

La variable se define como:

```
const TStage _stages_eyes[STAGE_COUNT]
```

donde STAGE_COUNT es el número de stages y que está definido como sigue:

```
#define STAGE_COUNT 256
```

Para generar automáticamente el fichero de cabecera construimos un sencillo programa en C que se encargue de leer el fichero xml y escribir los datos en el fichero .h. Este programa utiliza las librerías libxml2 de gnome [http://xmlsoft.org/] y la librería stardard de C++ ostream para manejar los ficheros.

Los parámetros de umbral alpha y theta_b encapsulados en el fichero XML son de tipo punto flotante. Este tipo de datos requieren de una gran cantidad de recursos para su procesado. Por lo tanto se decide hacer una conversión de tipos. El programa desarrollado es capaz de calcular el valor máximo y mínimo de los valores almacenados para el parámetro alpha. Se ha verificado que todos los valores están comprendidos en el rango [-5.0 - 5,0].

A continuación se ha tomado la decisión de mapear estos valores utilizando enteros de 16 bits. Se mapea el rango de valores float [-10.0 – 10.0] (doble del rango detectado) en el rango de enteros (int) [-30000 – 30000]. El programa desarrollado se encarga de realizar esta operación para cada término leído en el fichero XML antes de almacenarlo en el fichero de salida. Al realizar esta conversión, se produce un ligero sesgo en los datos alpha y theta_b. Con posterioridad comprobaremos que tras este ajuste, los resultados del detector no se ven afectados significativamente y se obtiene el mismo resultado en todas las pruebas realizadas.

Finalmente cada uno de los ítems de la estructura creada y encapsulada contiene 8 elementos de los cuales uno es un array de 17 valores. En total son 60bytes por “stage”. El tamaño total de la ROM que será necesario implementar será, por tanto, de 60 x 256 = 15.360 Bytes.

4.2.2. Algoritmo de detección de ojos

El algoritmo de detección es de tipo WaldBoost, que utiliza extractores de características de tipo Local Rank Differences (LRD). Este tipo de algoritmo ha sido ya presentado en el apartado de Estado del Arte (fig 7).

El operador LRD que se utiliza en el proyecto se caracteriza por trabajar con cuatro tipos diferentes de patterns. Estos patterns resultan de la combinación de matrices de tres por tres celdas donde cada celda puede estar compuesta por uno o dos pixeles en la dimensión ‘x’ (o ‘columna’) y uno o dos pixeles en la dimensión ‘y’ (o ‘fila’). Resultan por lo tanto cuatro posibilidades que se muestran en la siguiente figura:

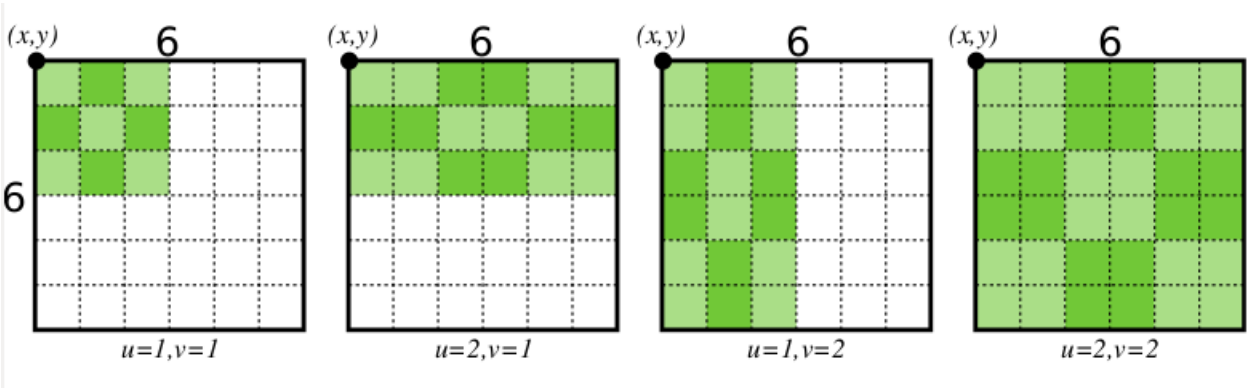


fig 13: Diferentes patterns para la extracción de características con LDR

Para extraer la característica utilizando el operador LRD:

a) En primer lugar determinaríamos el “stage” o nivel actual.

b) A continuación ubicaremos la posición, dentro de la ventana de muestreo (25x25 pixeles), en donde vamos a extraer la característica (cuadrado verde), con los parámetros x e y.

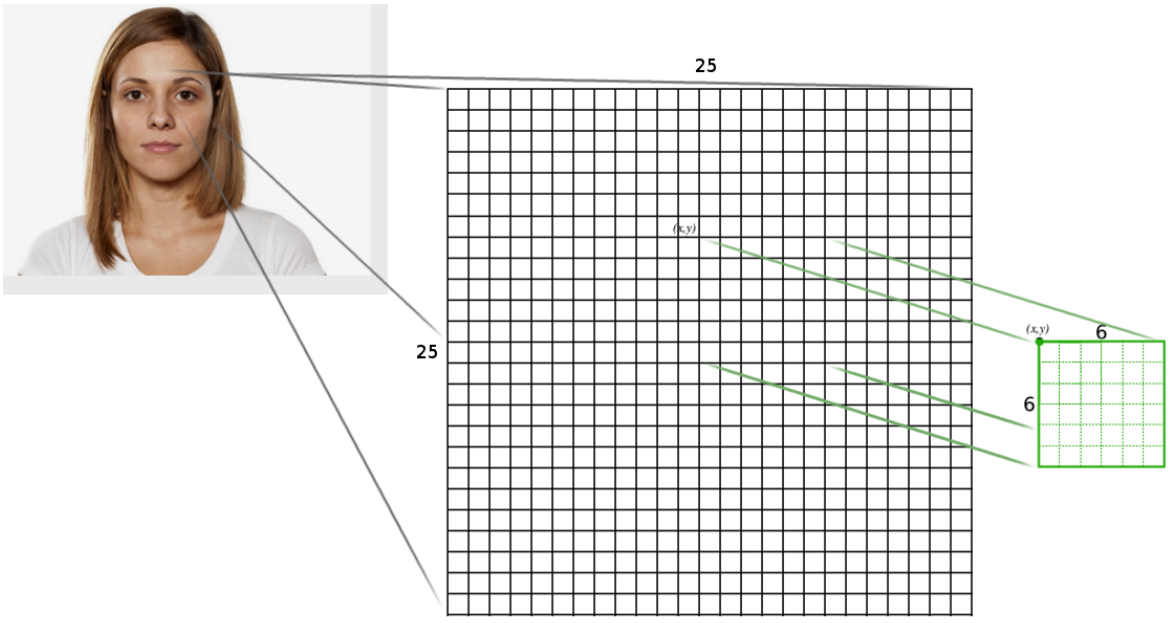


fig 14: ubicación de la característica dentro de la muestra de la imagen

Los parámetros x e y (13 y 8 respectivamente en el ejemplo de la figura 14) representan la posición dentro de la ventana de muestreo donde vamos a extraer la característica correspondiente al “stage” actual. Como puede verse en la figura 13, el pattern de mayor tamaño que puede tener un caracterizador LDR en este detector es una matriz de 6x6 pixeles. Por lo tanto, extraeremos la matriz con los 6x6 pixeles con coordenadas ‘x’ e ‘y’.

c) Determinaremos el tamaño de la característica con los parámetros w y h. Estos parámetros indican el número de pixeles que constituyen cada celda de la matriz final 3x3 que utilizaremos para realizar la operación LDR. Tanto w como h pueden tener para este detector concreto únicamente dos valores posibles: 1 ó 2, dando lugar a los cuatro posibles patterns mostrados en la figura 13.

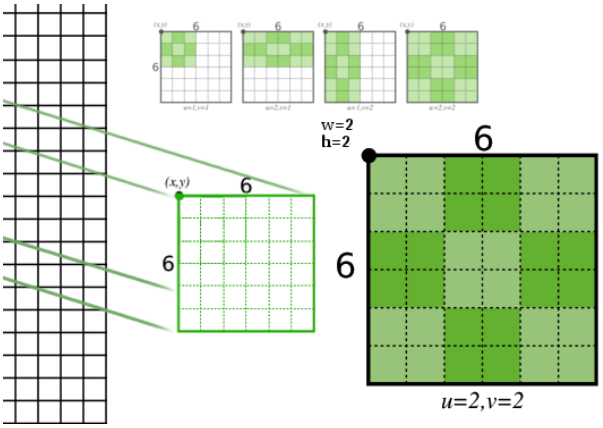


fig 15: selección del tamaño del feature (en el ejemplo: clusters de 2x2)

d) La operación LDR se realiza sobre una matriz de 3x3 pixeles. Cada elemento de la matriz de 3x3 se obtiene sumando los píxeles de cada agrupación (cluster) del pattern (celdas 2x2 del mismo color en la figura 15). Como la obtención del “Rank” se realiza con comparaciones entre pixels de la matriz, no es necesario realizar la media ya que el resultado es el mismo

e) y de este modo ahorramos las operaciones de división. De este modo, si el valor de h=1 y el de w=1, el cluster estará formado por un solo píxel y la matriz se formaría con la primera matriz 3x3 a partir del pixel con coordenadas x e y. Si h=2 o w=2, cada cluster estará formado por dos pixeles contiguos en el eje vertical u horizontal respectivamente, y cada celda de la matriz final 3x3 se calculará como la suma de los dos pixels de cada cluster. En el caso de que

tanto h como w sean igual a 2 el tamaño de cada cluster serán 4 pixels que habrá que sumar igualmente para obtener cada celda de la matriz 3x3 final. En el ejemplo podemos ver este último caso y como cada cluster de 2x2 pixels (con el mismo tono en la matriz 6x6 de la imagen 16) se suman para formar el valor correspondiente en la matriz a operar.

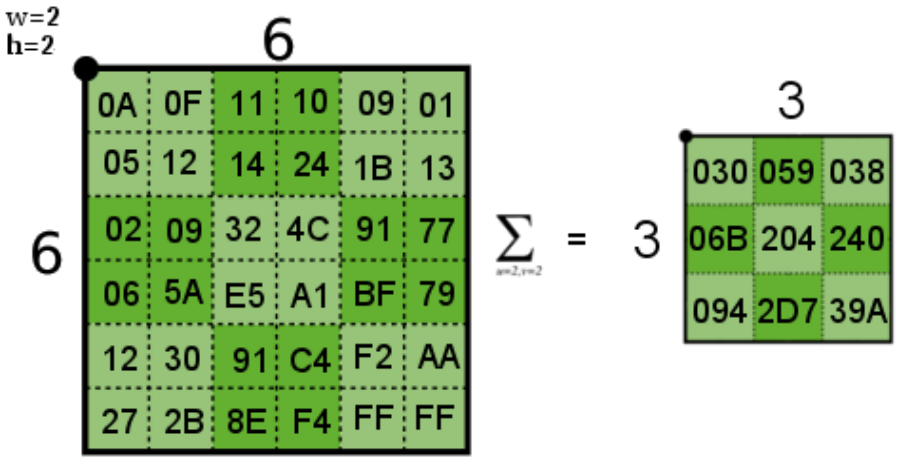


fig 16: Ejemplo de reducción de matriz 6x6 a 3x3 para w=2 y h=2

f) Obtendremos los valores de comparación para los rangos leyendo los pixels indicados por los parámetros A y B. En este ejemplo A vale 2 y B vale 6, por lo que leeremos los valores 059 para el pixel A y 240 para el pixel B.

g) Obtenemos ambos rangos. Para ello compararemos el valor del pixel en la posición A (el 2º en el ejemplo) con cada uno de los demás. El valor del 'rank' para ese pixel será igual al número de celdas de la matriz cuyo valor sea menor En el ejemplo podemos ver como el valor de la segunda celda es menor que seis de los pixeles que componen la matriz. Su rango (Rank) será 6. El valor para el 'rank' asociado al pixel marcado por el parámetro B se calcula de igual modo. El pixel marcado como B (el sexto en el ejemplo, con valor 240) tiene un rango (Rank) igual a 2.

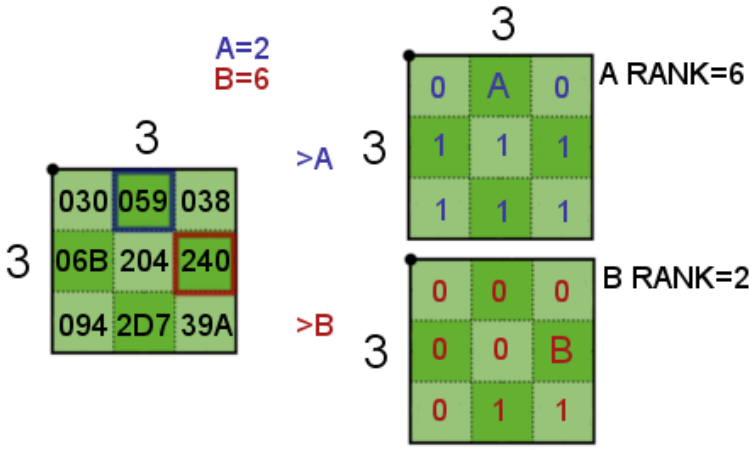


fig 17: Cálculo de los rangos de la imagen

h) Restamos y centramos el resultado en 8 para que el resultado sea positivo. Es decir, convertimos un valor entre -8 y 8 en un valor entre 0 y 16

   En el ejemplo anterior el resultado será de 12.

   A Rank – B Rank + 8 = 6 – 2 + 8 = 12

i) Utilizamos esta diferencia como índice para obtener el resultado del feature. El resultado coincidirá con el valor de la característica alpha cuyo índice corresponda con el obtenido en el paso anterior.

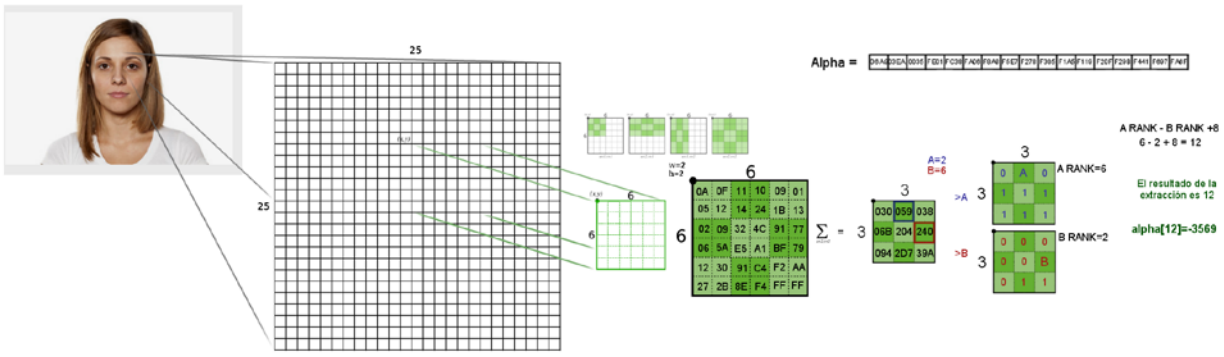


fig 18: Ejemplo de extracción de una característica LRD

Codificamos el algoritmo propuesto en lenguaje C utilizando las siguientes funciones:

```
short evalLRDStage (bw_img * image, unsigned smpOffset, void * s){
TStage * stg = (Tstage*) s; //leemos los parámetros del stage actual
int values[9] = {0,0,0,0,0,0,0,0,0};//pixels del feature
int * data = values; //puntero a values

// Localizamos en feature en la imagen
unsigned char * base = (unsigned char*)(image->data + smpOffset //
+ (stg->y * image->columnas + stg->x));

//obtenemos la matriz 3x3
sumRegions3x3(base, stg->w, stg->h, image->columnas, values);

int countA = 0;
int countB = 0;
int valA = values[stg->A];
int valB = values[stg->B];

// calculamos el rango
while (data < values + 9) {
if (valA > *data) ++countA;
```

código 2: Función que evalúa un feature LDR sobre una imagen

```
void sumRegions3x3(unsigned char * data,
int w, int h, unsigned width, int * v){
unsigned int blockStep = h * width;
Int x, y, i;

// Inicializamos el array
unsigned char * base[9] = {
data, data+w, data+2*w,
data+blockStep,data+blockStep+w,data+blockStep+2*w,
data+2*blockStep,data+2*blockStep+w,data+2*blockStep+2*w};

for (y = 0; y < h; y++) {   //Por cada h
for (x = 0; x < w; x++){    // Por cada w
for (i = 0; i < 9; ++i){    //Por cada pixel
v[i] += *base[i];
```

código 3: Función que obtiene una matriz 3x3 a partir de una imagen y los parámetros propios del stage

El valor devuelto por la función evalLDRStage ha de ser acumulado sucesivamente para cada uno de los 256 stages y comparado recursivamente con el valor theta_b asociado a cada uno de ellos. En el momento en el que una hipótesis no se cumpla (el acumulador sea menor que theta_b) estimamos que no se ha producido detección y desplazamos la ventana de muestreo.

El algoritmo ha de desarrollarse en tres dimensiones:

- La dimensión filas
- La dimensión columnas
- La dimensión stages (256 iteraciones)

El procesado descrito estará formado por un triple bucle.

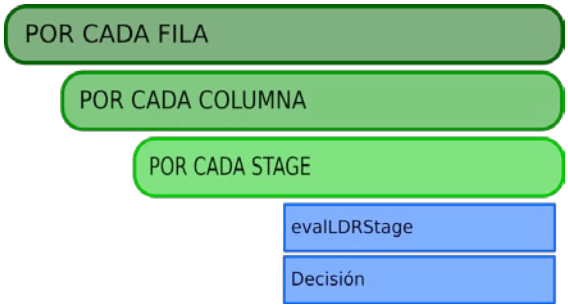


fig 19: Secuencia y dimensiones de procesado

Para poder comprobar el funcionamiento del código utilizamos la librería openCV[23][24] para capturar de manera sencilla la imagen de entrada, realizar preprocesado si fuese necesario, mostrar los resultados de manera gráfica e incluso utilizar una webcam como dispositivo de entrada.

fig 20: Salida del programa utilizado para probar el algoritmo en el PC

Tras esta fase de diseño hemos conseguido una descripción funcional ejecutable del algoritmo que es capaz de trabajar en un PC. Se realizaron varias pruebas y finalmente se consideró validada la funcionalidad del detector y del algoritmo en un PC.

4.2.3. Desarrollo de la plataforma hardware de base

Una vez desarrollado y probado el algoritmo necesitamos disponer de una plataforma base para su implementación en FPGA. Esta plataforma se construirá sobre el tandem ZedBoard con la tarjeta de adquisición de video HDMI AES-FMC-HDMI-CAM-G.

Para esta tarea intentaremos no tener que desarrollar ningún tipo de hardware propio para acelerar el proceso y utilizaremos en la medida de lo posible IP's del repositorio de Xilinx. Las interfaces de entrada y salida de video HDMI requerirán un bloque de adaptación entre los conectores HDMI y un bus del sistema a diseñar. Además será necesario el código para la detección/inserción de la señal de sincronismo integrada en las señales HDMI. Estos bloques están disponibles como soporte de la placa de AVNET en su página web[25]: AVNET-HDMI-IN y AVNET-HDMI-OUT

Al disponer, a priori, de todos los bloques funcionales ceñiremos el trabajo a la integración de IP utilizando "Vivado IP Integrator" para definir el sistema.

El objetivo será desarrollar un "PassThrougth" de la señal de video a través de la placa de desarrollo desde la entrada hasta la salida HDMI. Posteriormente podremos utilizar este proyecto como base para añadir el procesado de video propio a lo largo del canal de video creado.

Para este sistema podemos optar por tres estrategias diferentes [26]:

Arquitectura 1. Generar un stream de video directo entre la entrada y la salida:

Es la arquitectura más sencilla, pero es imprescindible que todos los bloques de procesado sean capaces de trabajar en tiempo real. Por ello es una arquitectura crítica en cuanto al tiempo de procesado

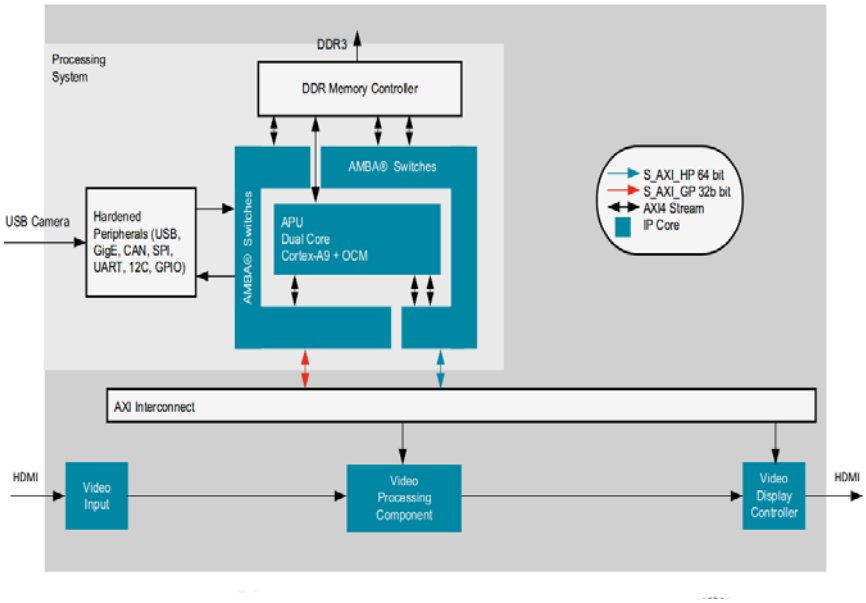


fig 21: Estructura de una arquitectura "Direct Streaming" según [26]

Arquitectura 2. Entrada y salida aisladas a través de un Frame Buffer en memoria externa:

En este caso, los bloques de procesado pueden trabajar directamente sobre el stream de entrada o sobre el stream de salida. Sin embargo, la sincronización entrada/salida se simplifica gracias al buffer intermedio. Esta arquitectura posibilita el acceso a los frames por parte de otros componentes del sistema como por ejemplo la CPU

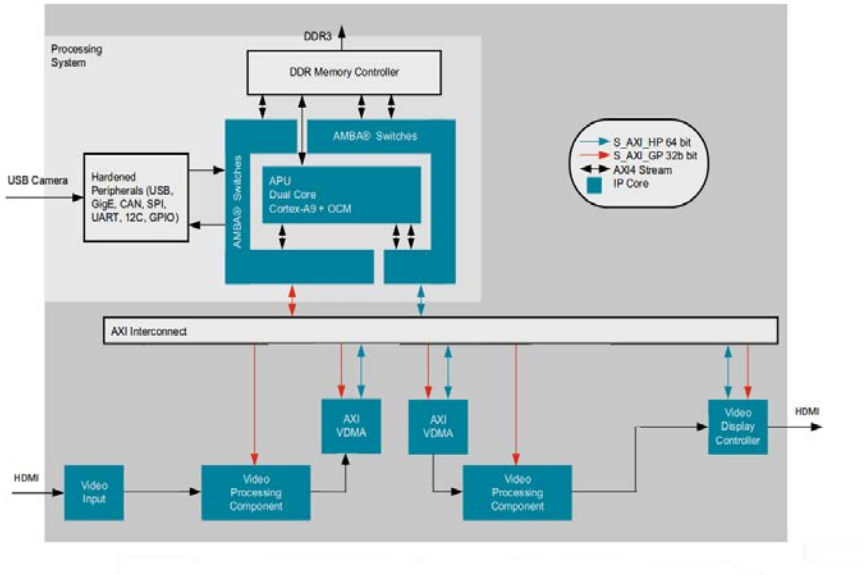


fig 22: Entrada y salida separada mediante un Frame Buffer.

Arquitectura 3. Componentes de procesado aislados entre dos Frame Buffer:

Esta es la arquitectura recomendada por Xilinx. En este caso aislamos los bloques de procesado y los hacemos insensibles al retraso del streaming de entrada salida.

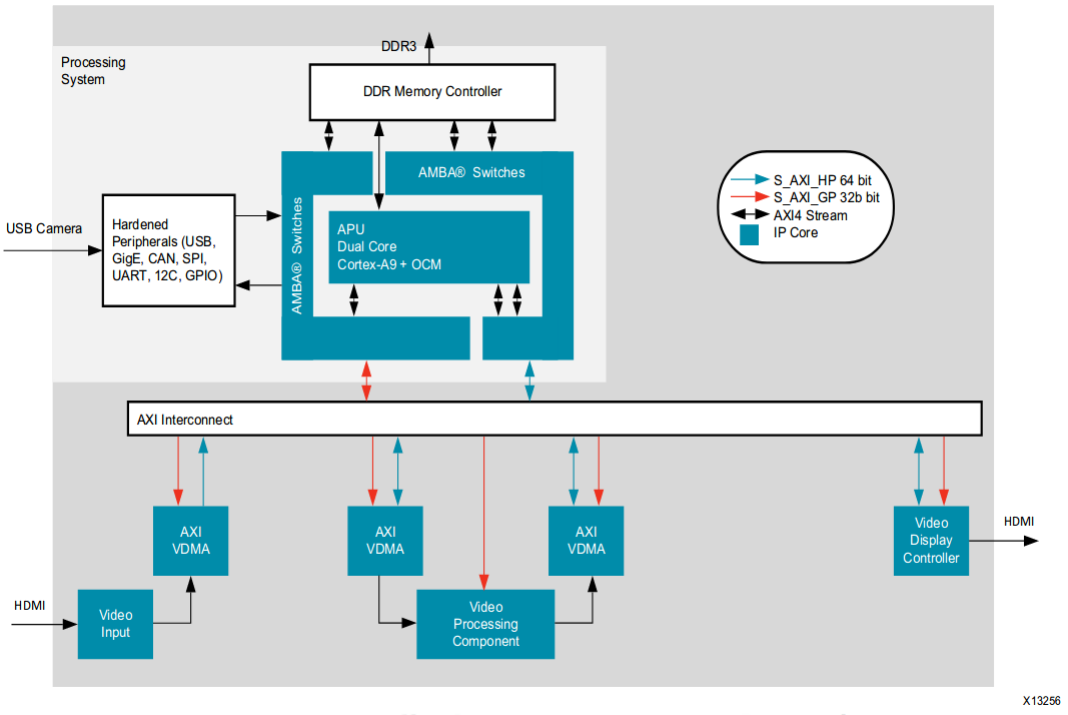


fig 23: Estructura de una arquitectura basada en "Frame Buffers" según [26]

Idealmente nuestro sistema debería ser capaz de trabajar en tiempo real y cumplir las especificaciones de la primera arquitectura. Para la plataforma base, inicialmente nos decantamos por una arquitectura basada en un único Frame Buffer con el componente de procesado ligado directamente al stream de entrada de video. Al no ser el objetivo de nuestro sistema producir una salida de video procesada, sino que simplemente obtendremos las coordenadas de los objetos encontrados, esta arquitectura nos permitiría obtener resultados en tiempo real.

Aunque trabajamos en las primeras fases del desarrollo siguiendo esta arquitectura, en el momento en que sintetizamos las primeras versiones del detector comprobamos que no resulta sencillo alcanzar los requerimientos de tiempo (latencia, initialization interval, pixel frecuency, etc). Por lo tanto, se abandonó el modelo anteriormente descrito y se adaptó una arquitectura en la que el bloque creado se encuentre conectado entre dos VDMA (Video DMA) que proporcionan sendos Frame Buffer residiendo en la memoria DDR RAM externa (arquitectura 3).

Acorde a este modelo desarrollamos y proponemos el siguiente sistema:

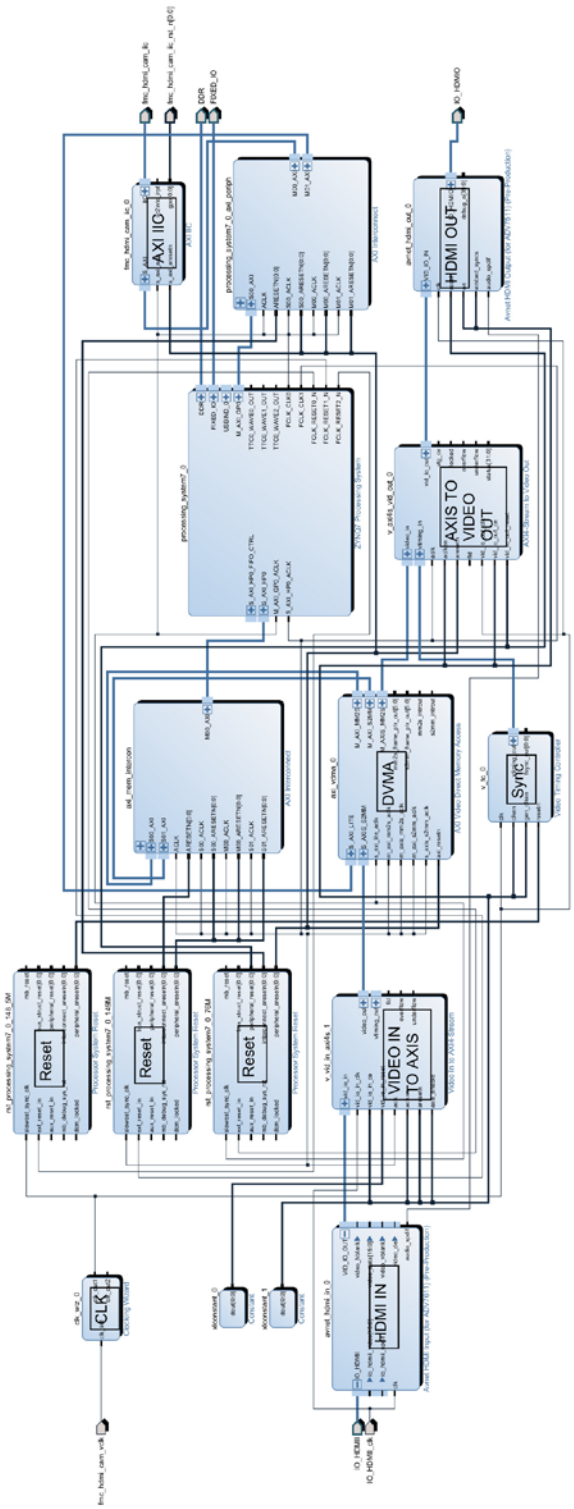


fig 24: propuesta de diagrama de bloques siguiendo la estrategia de la figura 22

El nuevo sistema se ciñe al esquema siguiente:

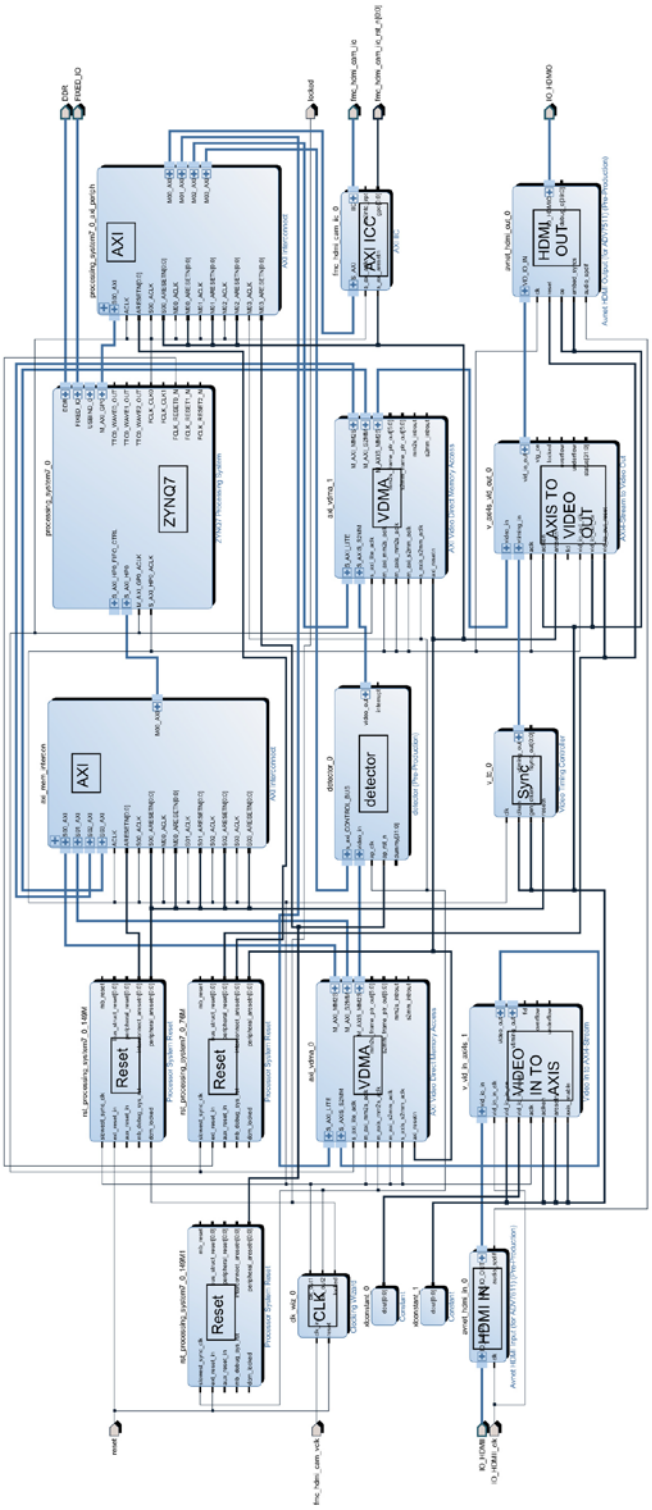


fig 25: Diagrama de bloques definitivo

El sistema base está compuesto por los siguientes IP's :

Avnet HDMI input (for ADV7611)	Extrae las señales de sincronismo (Vsync, Hsync, Vblanck, Hblanck) intrinsecas en la trama de entrada HDMI.
Video-In to AXI4-Stream	Convierte una señal de video estandar en un stream de datos de tipo AXI4-Stream [27]
AXI Video Direct Memory Access	Permite crear un Frame Buffer en la RAM externa. Dispone de una interfaz de lectura y otra de escritura. Recive/Envía el frame a través de una interfaz AXI4-Stream [28]
Clocking Wizard	Permite ajustar señales de reloj para su uso en el sistema. [29]
Video Timming Controller	Permite crear señales de sincronismo para video. [30]
AXI4-Stream to Video-Out	Permite construir una interfaz de video síncrona a partir de un stream de video y una señal de sincronismo. [31]
Avnet HDMI video out (for ADV7511)	Construye la trama de salida HDMI integrando las señales de sincronismo (Vsync, Hsync, Vblanck, Hblanck).
AXI IIC	Bridge AXI4-Lite I^2C. Nos permite enviar y recibir tramas del bus desde la CPU [32]
AXI Interconect	Permite enlazar varias interfaces AXI. [33]
Processor system Reset	Permite generar y controlar señales de reset de distinta naturaleza (sincronas, asincronas, negativas,, etc) [34]
Zynq7 Processing Sistem	Es el IP que da acceso a las interfaces del PS del chip Zynq7.[35]

El esquema funcional es el siguiente:

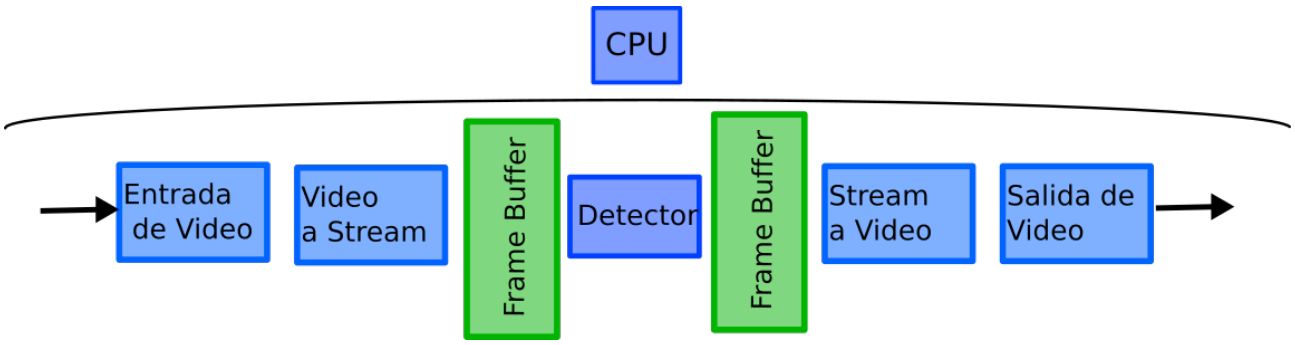


fig 26: Esquema funcional del circuito

El circuito se divide en varios bloques lógicos. Los bloques "Entrada de Video" y "Salida de Video" se conectan al conector FMC (VITA 57.1 FPGA Mezzanine Card) de la ZedBoard y a través de él con el receptor (ADV7511) y el transmisor (ADV7611) HDMI que se encuentran en la placa AES-FMC-HDMI-CAM-G.

Dentro de la FPGA, cada bloque lógico se enlaza en cascada con el siguiente tal y como se muestra en el diagrama de la figura 26 para formar una cadena de procesado de la señal de video entre la entrada y la salida.

El bloque lógico que denominamos CPU en el esquema se encargará de las conexiones propias con el procesador, la memoria, así como las interfaces I2C que utilizaremos para la configuración de los elementos que se encuentran en la placa HDMI (AES-FMC-HDMI-CAM-G) como la interfaz UART que utilizamos para la interfaz de usuario.

A continuación vamos a describir estos bloques funcionales y como es la relación de estos con la electrónica que se encuentra en la placa AES-FMC-HDMI-CAM-G:

4.2.3.1. **Entrada de video**

Recibiremos la señal de video proveniente de la tarjeta de entrada/salida de video HDMI de Avnet. Dicha tarjeta tiene el siguiente esquema:

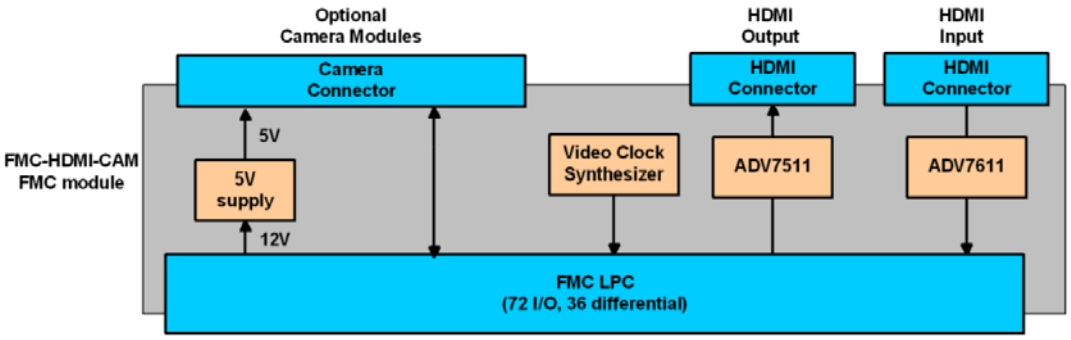


fig 27: Diagrama funcional del hardware de la placa AES-FMC-HDMI-CAM-G

El conector FMC (VITA 57.1 FPGA Mezzanine Card) de esta placa se conecta directamente al bloque PL (Prograable Logic) del Zynq7 de la ZedBoard donde residirá el resto del diseño implementado en la FPGA. La placa incluye un chip (AVD76118383) que decodifica la señal HDMI. En la siguiente figura podemos ver un esquema de las conexiones de este chip dentro de la placa AES-FMC-HDMI-CAM-G.

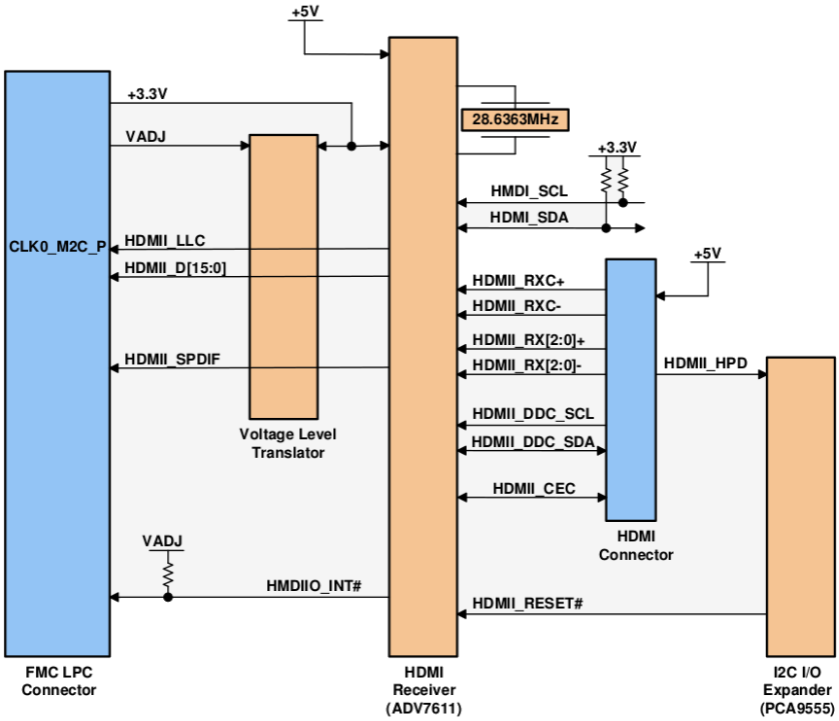


fig 28: Esquemático de entrada HDMI en AES-FMC-HDMI-CAM-G

Dentro de la FPGA, el IP “Avnet HDMI Input” acondicionará la señal y extraerá las señales de sincronismo: hblank y vblank . Utilizaremos el reloj de referencia de la placa FMC de 150Mhz para sincronizar este IP.

La configuración del chip ADV7611 se realizará a través de su interfaz I2C también disponible en el conector FMC. El Bus I2C es compartido por todos los dispositivos de la placa a través del multiplexor PCA9548A tal y como se muestra en la figura 29. Para el control del bus I2C se conectan los pines SDA y SCL del conector FMC al IP “AXI IIC” en la FPGA, que permitirá leer y escribir en el chip externo utilizando una interfaz AXI-Lite. La interfaz AXI-Lite se maneja como una memoria. Podremos controlar los dispositivos escribiendo o leyendo de los registros correspondientes mapeados en la estructura general de memoria del procesador.

El formato de video de entrada será de tipo YUV422 y por lo tanto la anchura del bus será de 16 bits: 8bits para la componentes luma y 8 bits para las componentes u y v de croma que se recibirán intercaladas en pixels alternos.

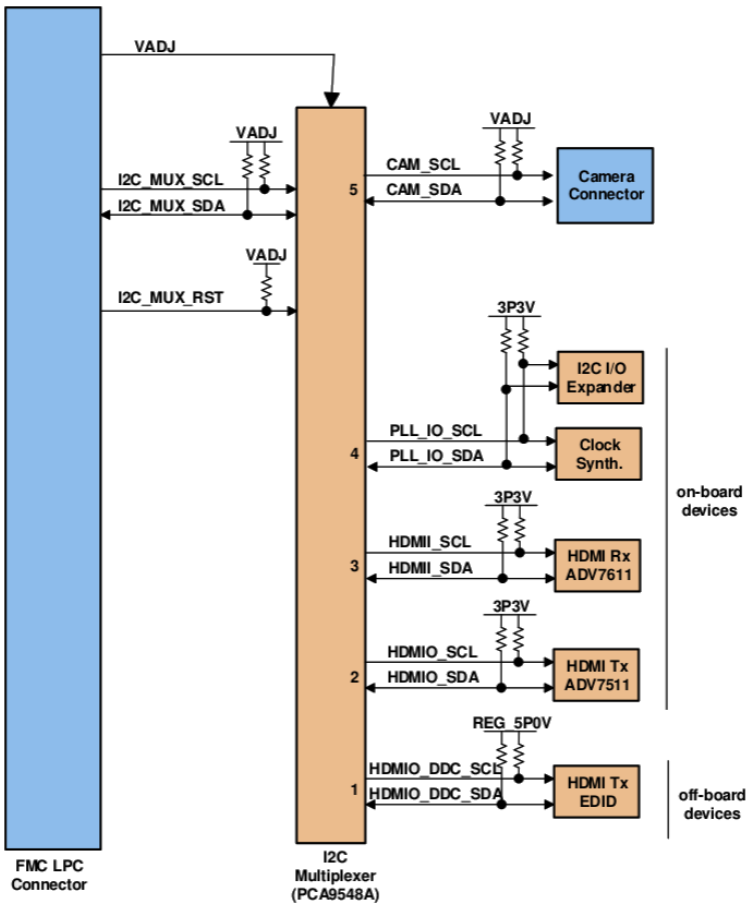


fig 29: Esquematico del circuito de multiplexado del bus I2C en AES-FMC-HDMI-CAM-G

A través del conector FMC recibiremos también una señal de reloj. Esta señal, la utilizaremos para excitar la mayor parte del circuito dentro de la FPGA. La señal de reloj se genera a través de un sintetizador de reloj CDCE913 que ha de ser configurado e inicializado, como ya mencionamos con anterioridad, utilizando una interfaz I2C. La frecuencia a la que trabajaremos será 150Mhz.

Para poder distribuir mejor la señal de reloj utilizaremos el IP “Clocking Wizard” que estabiliza la señal y añade buffers y drivers para aumentar su fan-out.

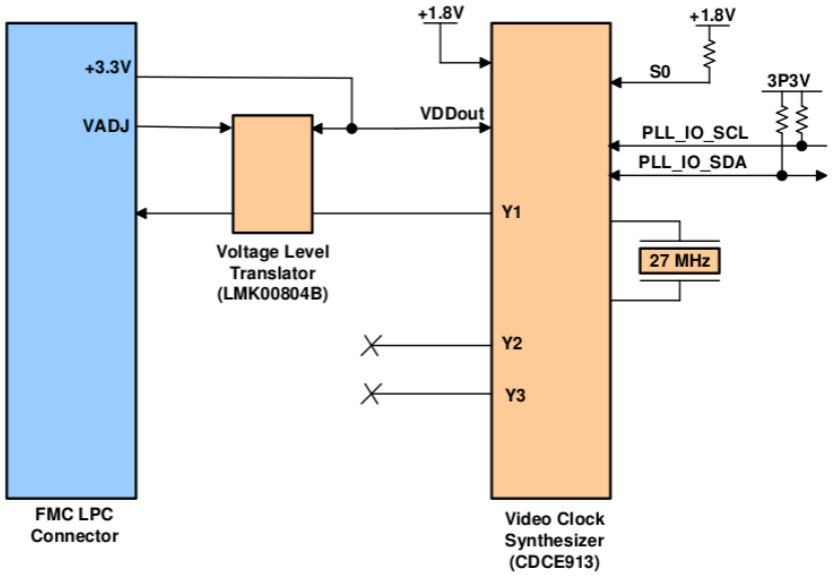


fig 30: Esquemático del "Video Clock Synthesizer"

4.2.3.2. **Video a Stream.**

De la conversión de la señal de video generada por “Avnet Video Input” se encarga el IP core “Video in to AXI4-Stream”. Este IP disponible en los repositorios de Xilinx (LogiCORE) construye un Stream de video a su salida. El IP debe ser configurado acorde al tipo de señal de video que recibirá. Además del payload de video, y de las señales de sincronismo del protocolo AXI4-STREAM (Tvalid y Tready) incluye las señales Tlast para indicar el fin de linea (EOL) y Tuser para señalizar el comienzo de un nuevo frame (SOF).

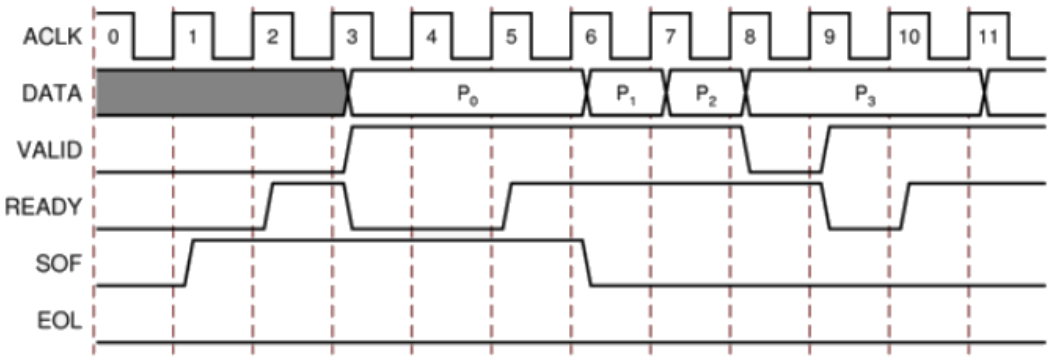


fig 31: Ejemplo de secuencia de sincronización de señales en AXI4 Video Stream

4.2.3.3. Frame Buffer

Los buffers de video se utilizan para aislar los bloques de procesado de video. Dichos bufferes están construidos en base a DMAs. Para implementar el acceso a memoria utilizamos los IP cores “AXI Video Direct Memory Access [28]” disponibles también en los repositorios Xilinx. Estos IP’s son muy versátiles y configurables, disponiendo de un canal AXI4-Stream de lectura y uno de escritura. El componente estará conectado al sistema DMA del Zynq7 (accesible desde el IP core “Zynq7 Processing System”) a través del bus principal AXI4 para el acceso a la DDR.

El VDMA dispone también de una interfaz AXI4-Lite para su configuración e inicialización en tiempo de ejecución. La sincronización entre los bufferes de entrada y salida (Frame Synchronization) se realiza a través de un sistema de buffer múltiple (triple buffering) que el VDMA es capaz de gestionar automáticamente.

4.2.3.4. Stream a Video

Esta es la etapa antagónica del módulo “Video a Stream”. Para esta tarea utilizaremos el IP core “AXI4-Stream to Video Out [31]” que es afín al “Video In to AXI4-Stream [27]”. En este caso es necesario introducir una señal de sincronismo de video ya que el Video Stream de entrada tal y como lo leemos del VDMA no dispone del ella.

Para generar estas señales de sincronismo utilizamos el IP core “Video Timming Controller [30]” que generará el patrón de sincronismo para la señal de salida.

El “Video Timing Controller” se configurará para generar las señales Vsync, Hsync, Vblank, Hblank y Active_video para una resolución de 1080p y con un tamaño total del Frame de 2200x1125.

El “Video Timing Controller” es configurable en tiempo de ejecución a través de una interfaz AXI-Lite, pero en nuestra implementación, de momento, esta opción está deshabilitada y el formato de salida de video estará prefijado.

4.2.3.5. Salida de video:

Para la salida de video a monitor necesitamos recomponer la señal HDMI y de ello se encarga el IP core “Avnet Video Out”. Después la señal saldrá de la FPGA a través del conector FMC para llegar al transmisor ADV7511 [37] que contiene la placa de Avnet y que se encargará de generar las señales diferenciales TMDS de la señal HDMI.

El driver ADV7511 se configura e inicializa en tiempo de ejecución a través del bus I2C (figura 27) al igual que el driver de entrada. Utiliza también, la misma referencia de 150Mhz generada en la placa FMC.

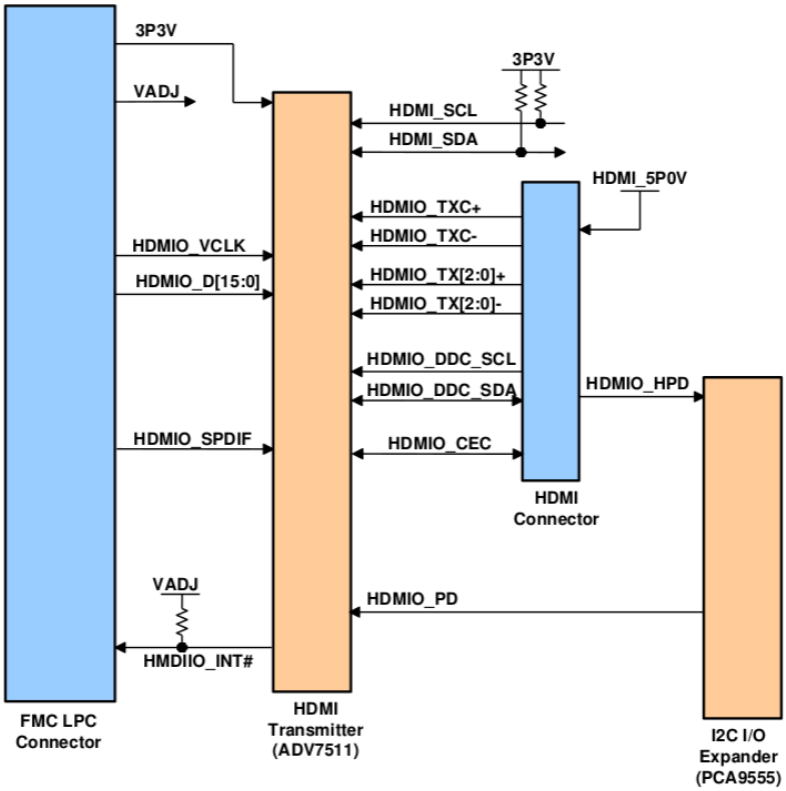


fig 32: Esquemático de salida HDMI en AES-FMC-HDMI-CAM-G

4.2.3.6. Detector:

Este bloque contendrá el circuito de procesado desarrollado en C y sintetizado medianta HLS. El nuevo módulo se conectará a la cadena existente, siendo necesario que sus interfaces de entrada y salida sean de tipo AXI4-Stream, con una anchura de bus de 16 bits (formato de video YUV422).

Se añadirá además una interfaz AXI4-Lite para en intercambio de información con el PS y para registrar las señales de proceso: start, done, ready e idle.

Por último, comentar que el circuito dispondrá de tres relojes:

FCLK_clk0: reloj generado por el Zynq7 de 100Mhz que controlará la lógica relacionada con los buses AXI-Lite y I2C.

clk_wiz_out0 y **clk_wiz_out1:** Tienen como referencia la señal de reloj de la placa AES-FMC-HDMI-CAM-G que se inyecta a través de el conector FMC. Estas señales tienen una frecuencia de 150Mhz y controlan el resto de componentes. La señal de reloj se propaga y divide utilizando el IP core “clocking wizard”, para facilitar el “place and route”.

Las señales de reset se controlan a través de los IP core “Processor system reset”. Añadiremos un IP de este tipo por cada reloj para poder sincronizar el sistema correctamente.

4.2.4. Desarrollo del Firmware/Software

Una vez generado el hardware del sistema en la fase anterior, desarrollaremos el software que nos permita controlar el sistema. Para ello debemos desarrollar los drivers de cada uno de los elementos que lo requieran. La arquitectura software que construimos se ciñe a la siguiente arquitectura:

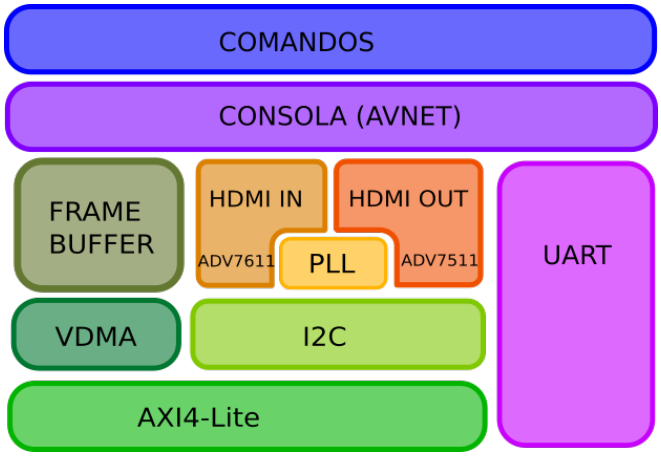


fig 33: Arquitectura del firmware de la plataforma base

El programa principal es muy sencillo y simplemente inicializa los dispositivos y cede el control a una consola de comandos de la interfaz de usuario. Se ha utilizado una consola desarrollada por Avnet sobre la interfaz UART de la ZedBoard, de modo que no tuvimos que codificar esta parte. La consola está desarrollada de manera modular y podemos añadir distintos comandos incluyendo una función que describa su comportamiento y unas pocas líneas de código más en el fichero “consola_avnet.cpp” con cabecera “consola_avnet.h”

El driver de más bajo nivel para la lectura/escritura en el bus AXI-Lite utiliza librerías que autogenera el SDK de Xilinx a través del módulo “Board Support Package” con librerías de sistema y drivers asociados a los IP cores incluidos en el diseño hardware. La misma situación se presenta para el VDMA.

Para el control del bus I2C que integra la placa Avnet AES-FMC-HDMI-CAM-G utilizamos el driver: *fmc_iic_sw_v2.03_a,* distribuido por Avnet

El sintetizador de reloj, el transmisor HDMI y el receptor HDMI también disponen de un driver distribuido por Avnet que se apoya en el anterior, para la lectura y escritura de los registros de la interfaz I2C de cada uno de los dispositivos: Este driver es: f*mc_hdmi_cam_sw_v2.01_a*

Con la totalidad de la base de software disponible, el desarrollo de la aplicación se redujo a la codificación de las funciones de inicialización haciendo las llamadas a las librerías descritas anteriormente. Se empaquetaron estas funciones en el módulo “demo.cpp” con cabecera “demo.h”. El programa principal, como hemos dicho, simplemente llama a estas rutinas y comprueba la entrada de datos de consola.

En la plataforma base, la interfaz de usuario se reduce a tres simples comandos:

start – que permite reiniciar el sistemas

verbose – que habilita o deshabilita la información de inicialización

help – que dá información sobre los comandos disponibles.

4.2.5. Implementación Hardware del algoritmo de detección

Una vez que disponemos de un sistema hardware/software base, iniciamos el desarrollo del algoritmo de reconocimiento.

El objetivo es llegar a alcanzar un sistema que sea capaz de trabajar con un alto frame rate sobre un Stream de video. Para poder tener un control más estrecho sobre este stream de video decidimos no utilizar las funciones de OpenCV.. Estas funciones se basan en estructuras Mat, propias de la librería OpenCV que hacen referencia a un frame completo.

En su lugar desarrollamos un buffer de entrada con el número de líneas necesario para el procesado de cada ventana de detección. El algoritmo, como ya hemos descrito, trabaja con ventanas de 25x25 pixels por lo que será necesario almacenar al menos 25 lineas para ir ejecutando el algoritmo.

Sin embargo, se ha observado que los “stages” del detector de ojos utilizan siempre coordenadas ‘y’ por encima de 20. Analizando los stages con detalle pudimos observar que el valor máximo de la coordenada ‘y’ sumando el tamaño en esta dimensión del feature (h) es siempre menor de 16.

De esta manera decidimos que para el detector de ojos concreto que implementaremos en este proyecto, el tamaño de la ventana sobre la imagen puede reducirse a 25x15.

Para comprobar este hecho, realizamos este cambio en el programa desarrollado en PC y comparamos el resultado sobre una docena de fotografías con el resultado obtenido utilizando una ventana de 25x25 y comprobamos que no hay ninguna diferencia.

Los detectores de que disponemos en el laboratorio han sido entrenados utilizando el algoritmo WaldBoost sobre ventanas cuadradas. La geometría rectangular del ojo (de una proporción aproximada a 5/3) hace que todas las características significativas que selecciona el algoritmo se encuentren en los 3 primeros quintos de la imagen, lo que justifica el uso de una ventana rectangular.

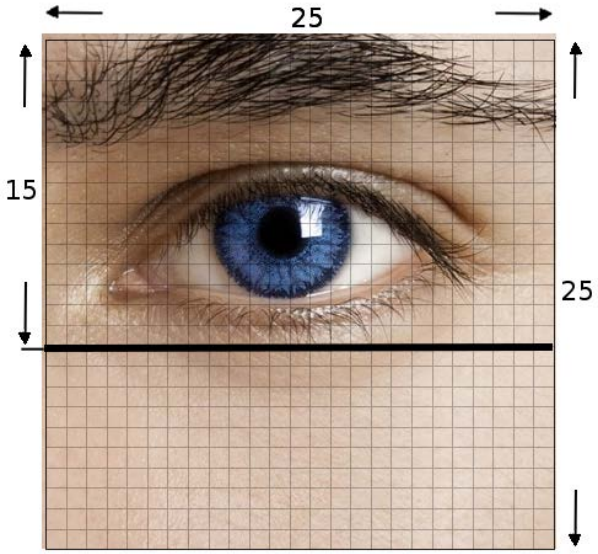


fig 34: Ajustamos la ventana útil a una matriz de 25x15

Por lo tanto, el primer paso en el desarrollo del detector fue crear un buffer de entrada de 15 lineas. Esta parte del algoritmo no era necesario en el programa de ordenador ya que en el PC almacenamos frames completos en memoria y trabajamos leyendo de manera más o menos aleatoria sobre una gran matriz del tamaño total de la imagen a procesar.

Para el diseño del buffer de entrada creamos dos variables.

a) **stage**: Una matriz de 25x15 pixeles que almacenará la ventana correspondiente a la coordenada actual dentro de la imagen.

b) **buffer**: Una matriz de 15x2200 pixels capaz de almacenar los pixels no utilizados en el stage actual de las últimas 15 líneas para poder desplazar la ventana.

Estas variables se definen como FIFOs (buffers circulares) para mejorar el rendimiento. La implementación de esta estructura es la siguiente:

```
//llenamos buffer
buff:for (n=0; n<(SAMPLE_HEIGHT-1); n++) {
#pragma HLS UNROLL

sample[n][sample_cursor]=buffer[n][buffer_cursor];
buffer[n][buffer_cursor]=sample[n+1][sample_cursor];
}
//ultima linea
sample[SAMPLE_HEIGHT-1][sample_cursor]=luma_in;

//desplazamos los apuntadores
buffer_cursor++;
if(buffer_cursor>(columnas-SAMPLE_WIDTH-1)) buffer_cursor=0;
sample_cursor++;
if(sample_cursor>SAMPLE_WIDTH-1) sample_cursor=0;
```

código 4: Ventana de procesado a partir del pixel actual. Buffer circular.

Después re-codificamos la función “*evalLRDStage”* que habíamos programado para PC (código 2) para trabajar sobre las estructuras previas en vez de utilizar apuntadores al grueso de la imagen. El resultado es una función: *feat_eval* que recibe como parámetros de entrada una matriz de 6x6 (el tamaño máximo del feature) y los parámetros para su procesado: Las posiciones de los pixels de referencia para el cálculo de los rangos (A y

```
unsigned feat_eval(short pre_feat[6][6], unsigned char tipe,
 unsigned A, unsigned B){
unsigned char l=0;
unsigned char u, v, h, w;
unsigned short feature[9]; //feature 3x3
#pragma HLS ARRAY_PARTITION variable=feature complete
unsigned short countA=0; //contador RANK A
unsigned short countB=0; //contador RANK B
#pragma HLS PIPELINE
//seleccionamos entre las cuatro posibles posibilidades de feature
//dependiendo de su tamaño que esta indicado por los parámetros w y h
switch (tipe){
case 0://3x3 -> w=1 h=1
case0u:for(u=0; u<3; u++){
case0v:for (v=0;v<3;v++){
feature[l]=pre_feat[u][v];
l++;}}
break;
case 1://6x3 -> w=1 h=2
case1u:for(u=0; u<6; u+=2){
case1v:for (v=0;v<3;v++){
feature[l]=pre_feat[u][v]+pre_feat[u+1][v];
l++;}}
break;
case 2://3x6 -> w=2 h=1
case2u:for(u=0; u<3; u++){
case2v:for (v=0;v<6;v+=2){
feature[l]=pre_feat[u][v]+pre_feat[u][v+1];
l++;}}
break;
case 3://6x6 -> w=2 h=2
case3u:for(u=0; u<6; u+=2){
case3v:for (v=0;v<6;v+=2){
feature[l]=pre_feat[u][v]+pre_feat[u][v+1]
+pre_feat[u+1][v]+pre_feat[u+1][v+1];
l++;}}
break;
default:
break;
};
eval:for(l=0; l<9; l++) {
//obtenemos los rangos
if(feature[A]>feature[l]) countA++;
if(feature[B]>feature[l]) countB++;
}
//devolvemos el indice
return (countA - countB + 8);}
```

código 5: Función que extrae las características del stage utilizando operador LDR

B) y el tamaño de los cluster de la matriz 3x3 sobre la que se realiza la operación LDR (w y h).

Además, se ha modificado el código basándonos en la propuesta de Zemčík, Juránek, Musil, Musil y Hradiš en [13] para una implementación hardware para un clasificador de tipo LRD

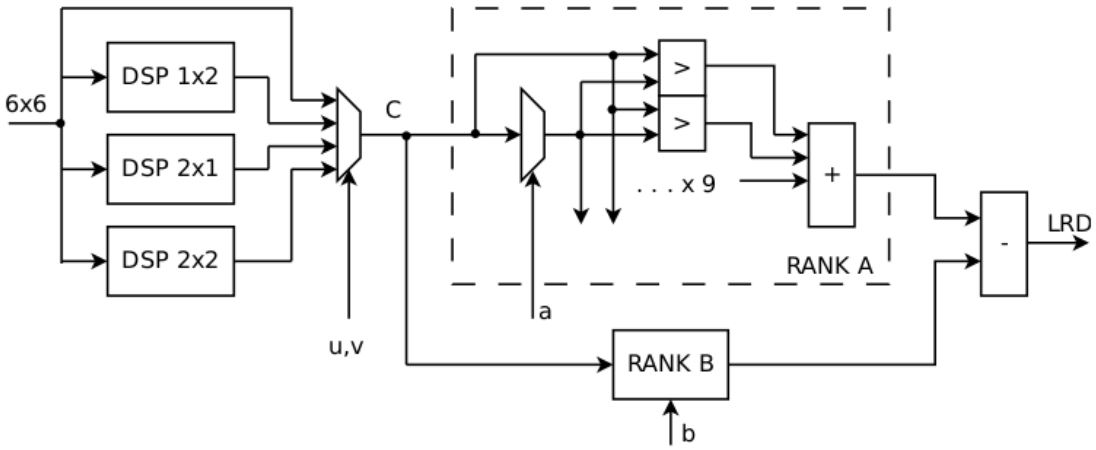


fig 35: Esquema de implementación de clasificador LRD hardware según [13]

Para codificar el sistema utilizamos una sentencia "*switch"* para implementar el multiplexor en función de u y v (w y h en nuestro caso). El resto serán sumas y comparaciones.

Los parámetros de cada stage del detector se implementan como una ROM. Para ello utilizamos, como ya explicamos en el apartado "Preparación y Encapsulado del Detector", una estructura que encapsula los datos necesarios y una variable del tipo de esta estructura con forma de matriz de 256 elementos preinicializados. Esta parte del código es descrita en un fichero: "_stages_eyes.h".

Para poder hacer la llamada a la función "feat_eval", previamente hacemos una lectura de los parámetros de la estructura correspondientes a cada stage desde la ROM. Después, conocidos 'x' e 'y' construiremos la matriz de 6x6 que pasaremos a la función realizando las lecturas de las posiciones indicadas en la matriz de 25x15 (stage)

```
//lectura de la ROM
s_x=_stages_eyes[s_act].x;
s_y=_stages_eyes[s_act].y;
A=_stages_eyes[s_act].A;
B=_stages_eyes[s_act].B;
//determinamos el tipo de feature 3x3, 3x6, 6x3, o 6x6
tipe=((_stages_eyes[s_act].w-1)*2)+(_stages_eyes[s_act].h-1);

//recogemos los datos del feature desde el marco
prefeaty:for(y=0; y<6; y++) {
prefeatx:for(x=0; x<6; x++) {
//la coordenada en x depende de la posición en el buffer circular
//debemos sumar el cursor y hacer el módulo con el tamaño total
s_x_calc=s_x+x+sample_cursor;
if(s_x_calc>=SAMPLE_WIDTH) s_x_calc-=SAMPLE_WIDTH;
pre_feat[x][y]=sample[s_y+y][s_x_calc];
}
}

//evaluamos el feature
sub_record=_stages_eyes[s_act].alpha[feat_eval(pre_feat, tipe, A, B)];
```

código 6: Extracción de matriz y procesado del feature

Para finalizar, comprobaremos si se cumple la condición de detección (comprobaremos las hipótesis) y, en caso positivo, registraremos los resultados en la memoria de salida.

El proceso de diseño, una vez desarrollado el código C, consistirá en simular y comprobar que el algoritmo es correcto funcionalmente. Después lo sintetizaremos y verificaremos los resultados post-síntesis donde nos fijaremos principalmente en la estimación de recursos consumidos en la FPGA y las estimaciones de tiempo de ejecución. Como los sistemas de síntesis de alto nivel generan diseños síncronos, las estimaciones de latencia y de los Intervalos de Reentrada son proporcionales en ciclos de reloj, lo que les confiere bastante precisión.

Por último, empaquetaremos el resultado como un IP core y lo exportaremos al proyecto base que habíamos desarrollado para poder implementarlo y probarlo.

4.2.6. Optimización durante la síntesis

Al igual que cuando desarrollamos software, al implementar hardware utilizando síntesis de alto nivel (HLS) es necesario trabajar concienzudamente la arquitectura del código para poder optimizar los resultados en cuanto a velocidad y consumo de recursos. En el caso de la síntesis HLS es quizá más crítico ya que las diferencias en el resultado dependen sobremanera de la estrategia y estructura del código C.

En nuestro caso, partimos de una codificación previa realizada para PC, que es un sistema secuencial en esencia

En el procesado secuencial, las CPU no tienen rival, funcionan con relojes por encima de 1Ghz. El reloj más rápido, según las especificaciones, que podemos utilizar en la FPGA de la ZedBoard es de 250Mhz.

Por lo tanto, un sistema que se ejecutara con el mismo nivel de paralelización, funcionaría mucho más rápido en una CPU que en una FPGA. La potencia del procesado hardware está en su capacidad de ejecutar tareas concurrentemente.

Añadido a las dificultades descritas, los sistemas basados en síntesis de alto nivel tienen una serie de limitaciones intrínsecas, como son el hecho de que solo utilizan un reloj, que son máquinas de estados completamente síncronas, etc. Son características que ayudan a la síntesis pero por otro lado es difícil conseguir resultados eficientes en tiempo de ejecución. Aunque el sintetizador tratará siempre de implementar sistemas de una manera eficiente no siempre va a ser posible que entienda y paralelice exactamente ‘que es lo que queremos’ y por eso es tan crítica la codificación.

A través de las directivas de síntesis podemos ayudar al sintetizador a comprender que estrategia de optimización llevar a cabo en cada segmento del programa e incluso que tipo de recursos hardware nos gustaría que emplease para elementos concretos.

Nuestra primera síntesis funcionalmente correcta (después de la simulación) que estaba codificada de una manera muy similar a como se escribió el programa para PC necesitaba entre 78 y 254 segundos para un frame VGA. Además, nunca conseguimos hacer funcionar esta implementación en la placa de desarrollo.

La naturaleza del algoritmo que nos ocupa presenta una serie de dificultades que se suman a las propias de la herramienta:

- El resultado de cada ‘stage’ depende del resultado de los anteriores. Esta dependencia hace que no podamos paralelizar multiplicando la lógica en esta dimensión ya que cada resultado debe esperar al anterior.
- Los patrones de procesado no son homogéneos, lo cual implica tener que añadir lógica de decisión que dificulta la paralelización.
- Los datos de entrada en cada ‘stage’ implican lecturas en posiciones muy aleatorias, lo cual penaliza mucho el tiempo de procesado incluso cuando no utilizamos memoria ya que la lógica de selección es muy compleja.

Como estrategias de optimización llevadas a cabo y que finalmente tuvieron resultados podemos destacar:

1. **El control sobre los mecanismos de almacenamiento para las distintas variables.**

El objetivo es reducir el número de accesos a memoria, o al menos, que estos pudiesen realizarse de manera simultánea. Para ello forzamos a utilizar registros en lugar de bloques BRAM siempre que sea posible. Si el buffer es muy grande y hemos de recurrir a memoria, trataremos de particionar y distribuir los datos de la manera más eficiente posible.

En algunas implementaciones, los recursos de memoria eran accedidos desde múltiples puntos del módulo lo que resultaba en que el sintetizador disponía demasiada lógica ligada a algunos elementos (bloques de memoria sobre todo). La síntesis de alto nivel ofrecía buenos resultados pero al implementarlo a nivel RTL se producían violaciones del tipo “timing constrains fail” debido a un tiempo de retraso muy elevado en la linea. En ocasiones, los retrasos por fanout llegaban a superar el periodo de reloj.

Para evitar esta problemática fue necesario revisar el código para que el flujo de datos estuviese mejor estructurado y distribuido por etapas, además de dividir las memorias para que los datos estuviesen también más y mejor distribuidos.

2. **Reestructuración de los bucles para evitar dependencias.**

Para que el sintetizador pueda optimizar los lazos es necesario que la descripción de los mismos tenga el menor número posible de dependencias y saltos no estructurados (break), además de reducir los accesos aleatorios a memoria.

Nuestro algoritmo tiene dos cuellos de botella importantes y difíciles de atajar en este sentido: por un lado la dependencia entre los resultados de los stages y por otro la condición de descarte. Ambos limitan y complican la optimización del bucle que recorre los stages.

3. **Utilizar una estrategia de implementación pipeline de lazos.**

Los intentos por desenrollar los lazos no ha resultado generalmente bien. En la mayoría de los casos el volumen de recursos necesario desbordaba la capacidad de la FPGA dado que el número de iteraciones es muy elevado. El desenrollado parcial tampoco ofrecía resultados suficientemente buenos.

Decidimos que por la naturaleza del algoritmo la estrategia más oportuna era paralelizar a través de pipelines o solapamiento de ejecución entre iteraciones.

Fue necesario reestructurar muy bien el código para conseguir un pipeline eficiente. Para ello es necesario atomizar las acciones a lo largo de la ejecución y diseñar un flujo de datos lineal y sin dependencias previas.

4. **Tratar de crear varios canales de procesado en las dimensiones fila/columna**

Una vez optimizado el proceso en la dimensión stage se trabaja en posibilidad de procesar varios pixels contíguos de forma paralela. El solapamiento de las ventanas de busqueda produce lecturas simultáneas que penalizan el tiempo o producen operaciones que desbordan el tiempo de un ciclo de reloj, por lo que se hace necesario de nuevo reestructurar la codificación.

Además, el número de recursos utilizados crece rápidamente, al incluir nuevas tareas que se ejecutan en paralelo por lo que el grado de optimización es bastante limitado.

4.2.7. Desarrollo del Driver.

El detector implementado recibe y envía los parámetros de entrada y salida de la función que lo describe a través de su interfaz AXI-Lite. Los parámetros de entrada que necesita para poder trabajar son: las filas y columnas**.** Estos parámetros son dos valores de tipo entero que determinan el tamaño de la imagen a procesar. El detector no es capaz de determinar el tamaño por si mismo y necesitamos que el programa lo inicialice.

El dispositivo ofrecerá los siguientes valores de salida:

1. **dummy:** es un contador que se incrementa en cada nuevo frame. Nos indicará el número de frames procesados. Además, leyendo su valor entre dos intervalos de tiempo determinados, podremos obtener el frame-rate del detector. El parámetro dummy es de tipo “unsigned int”.
2. **resultado:** Es una lista con las coordenadas de los objetos detectados en el último frame. Está declarado como un array de 25 filas en el que cada una estará compuesta por dos enteros cortos sin signo (unsigned short) correspondientes a las coordenadas ‘y’ y ‘x’ de cada objeto en la imágen. Solo se mostrarán los 25 primeros resultados positivos. Si en un frame hubiese más, a partir del resultado 25 no se registrarían a la salida.

Cuando sintetizamos un IP core utilizando HLS Vivado e incluimos interfaces AXI4-Lite, la propia herramienta genera, junto con el encapsulado del IP core, un driver software con las funciones básicas para la inicialización, el control de proceso y el intercambio de información a través de ese bus. Estas funciones se construyen tanto para su llamada desde aplicaciones corriendo sobre linux como para aplicaciones bare-metal (sin sistema operativo).

Además, el sistema crea funciones específicas para la lectura y escritura de cada uno de los registros. También para el manejo de las líneas de control: start, idle, ready y done.

Llamaremos a estas funciones desde la aplicación de la plataforma base que utilizamos para probar el detector:

Incluimos la configuración del detector en la función de inicialización: demo_init en demo.c

Los valores de entrada filas y columnas se leerán del driver hdmi de entrada ADV7611 a través del bus I2C. Cuando se obtengan estos valores al inicializar esté dispositivo, introduciremos los valores en el detector también y lo pondremos en marcha. Esta acción se realiza en la función demo_start_hdmi_in en demo.c

Además de esto crearemos una serie de comandos que permitan ejercer cierto control del detector para las pruebas. Los comandos desarrollados han sido:

a) **detstart**: este comando hará que el detector se ejecute de manera continua frame tras frame.

b) **detstep:** este comando realizará una única ejecución sobre el frame almacenado en el Frame Buffer.

c) **detstop:** detendrá la ejecución del detector al finalizar el frame actual.

d) **getdummy:** nos devolverá el valor de la variable dummy (contador de frames).

e) **getresult**: devolverá los valores de salida del detector al terminar la ejecución actual. El detector quedará detenido tras la lectura. En caso de que el detector estuviese detenido al ejecutar este comando, se iniciará la detección sobre un frame para poder obtener un resultado.

Todas las acciones que llevan a cabo estos comandos corresponden con una función específica por lo que las mismas acciones pueden llamarse desde otras secciones del código. Son estas funciones las que constituyen el driver del dispositivo.

5. Verificación

Los procesos de diseño y verificación son complementarios e indisolubles. Cada una de las etapas de diseño debe ser medida y contrastada con los requisitos y la funcionalidad esperada para conseguir que las especificaciones se cumplan en muchas ocasiones debemos hacer modificaciones en etapas anteriores y volver a verificar el diseño de nuevo. De tener una buena estrategia de verificación depende, en gran medida, la eficacia del proceso de diseño.

5.1. Verificación funcional del algoritmo

Cómo se ha descrito en el apartado de diseño, antes de implementar el sistema en FPGAdecidimos verificar y evaluar el funcionamiento del algoritmo desarrollando un programa para PC que nos permitiera comprobar su funcionamiento y medir su eficacia.

A partir de este código se han diseñado y llevado a cabo varias pruebas de validación.

Comprobación del funcionamiento: Se compilaron dos programas para verificar el funcionamiento del seleccionado y del código que describe el algoritmo.

En el primero de ellos se utiliza como entrada la señal de video de una webcam de resolución VGA capturada frame a frame con la clase VideoCapture de openCV.

El programa escalaba la imagen original recursivamente con factor $\sqrt{(2)}$(7/5) hasta alcanzar un tamaño menor a la ventana (25x25). Cada imagen escalada era procesada con el detector para obtener los resultados. Dichos resultados se filtran para evitar detecciones múltiples del mismo objeto y se devolvían las coordenadas de los objetos (ojos) detectados. La salida del programa era, además de las coordenadas mostradas en consola, la señal de video original visualizándose en una ventana donde sobre cada frame

se recuadraba los objetos detectados atendiendo a la escala en la que fueron encontrados. La figura 20 muestra la salida de uno de estos frames en este programa.

El segundo de los programas de prueba recogía como entrada una serie de imágenes con formato “.bmp” y las procesaba mostrando en consola el resultado de la detección aplicando sobre cada imagen el mismo escalado que en el programa anterior. Este programa también creaba una ventana con la representación gráfica de las detecciones sobre cada imagen. En la imagen 36 podemos ver un ejemplo de salida.

Podemos apreciar como las detecciones se producen en muchos casos en múltiples escalas. En la figura 37 podemos ver cómo además el mismo objeto se detecta en un rango variable de pixels anexos.

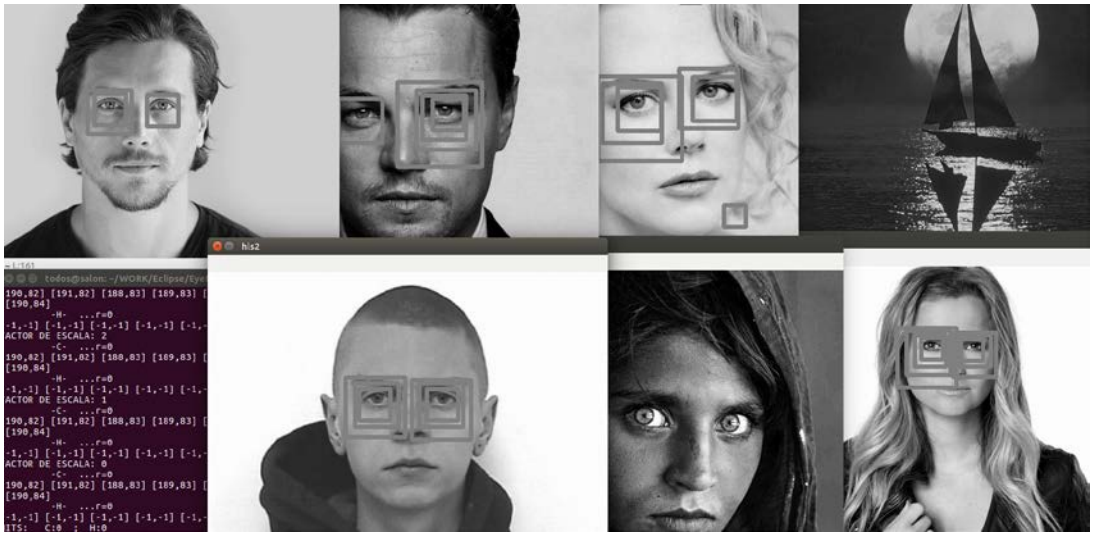


fig 36: salida del programa de prueba sobre PC para 7 fotografias de muestra de 640x480

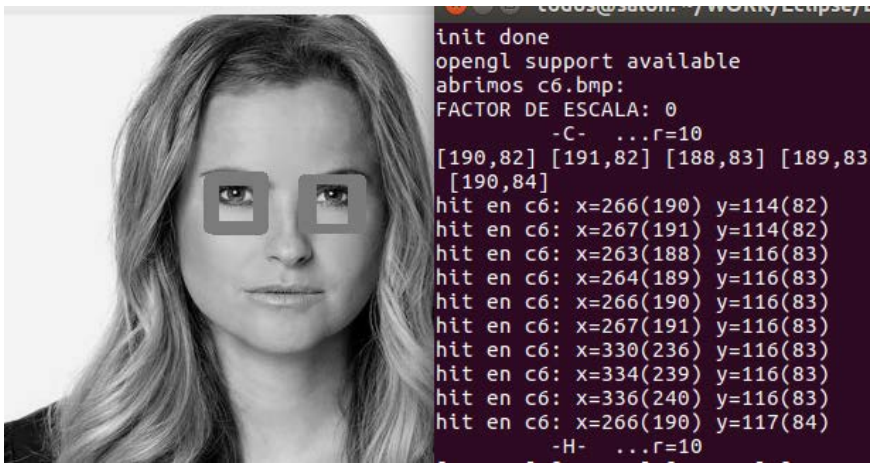


fig 37: salida del programa de prueba para una imagen de 457x343. En la consola podemos ver como se detecta el mismo ojo en varios pixels anexos

Para realizar estas pruebas se han recopilado diversas fotografías. La mayoría de las pruebas se ha realizado con una muestra de 20 fotografias de resolución VGA de caras frontales, 12 fotografías 1080p de imágenes frontales, 6 fotografías de resolución VGA sin caras y 6 fotografías 1080p sin caras (figura 38).

fig 38: Algunas de las imágenes de muestras utilizadas para las pruebas

Analizando los resultados sobre la muestra que hemos escogido observamos que los únicos ojos que nuestro detector no ha encontrado son los de la famosa imagen de national geographics de una niña afgana, quizá porque la imagen no es del todo frontal y el detector es muy sensible a este hecho.

Por otro lado, sobre algunas muestras (9 de las 44) se obtuvieron falsos positivos en el pelo y la nariz de algunas imágenes. Ningún falso positivo se produjo en las imágenes que no contienen caras.

La muestra no era grande y las fotografías han sido seleccionadas de manera bastante aleatoria a través de un buscador web pero el objetivo de este proyecto no era evaluar el detector sino verificar su funcionamiento.

Para intentar aportar más información sobre los resultados diseñamos otro experimento. Creamos un programa basado en los anteriores que generará un fichero CSV con el valor del stage en el que se ha descartado cada uno de los pixels. El resultado es una tabla de datos con un estudio estadístico de la detección en cada pixels. El resultado es curioso ya que la gran mayoría de los pixels se descartan en la primera iteración. Aquellos que no se descartan suelen coincidir con los bordes que pudiesen

detectarse en la imagen, con lo cual al observar los datos a distancia se puede apreciar la silueta de la imagen original.

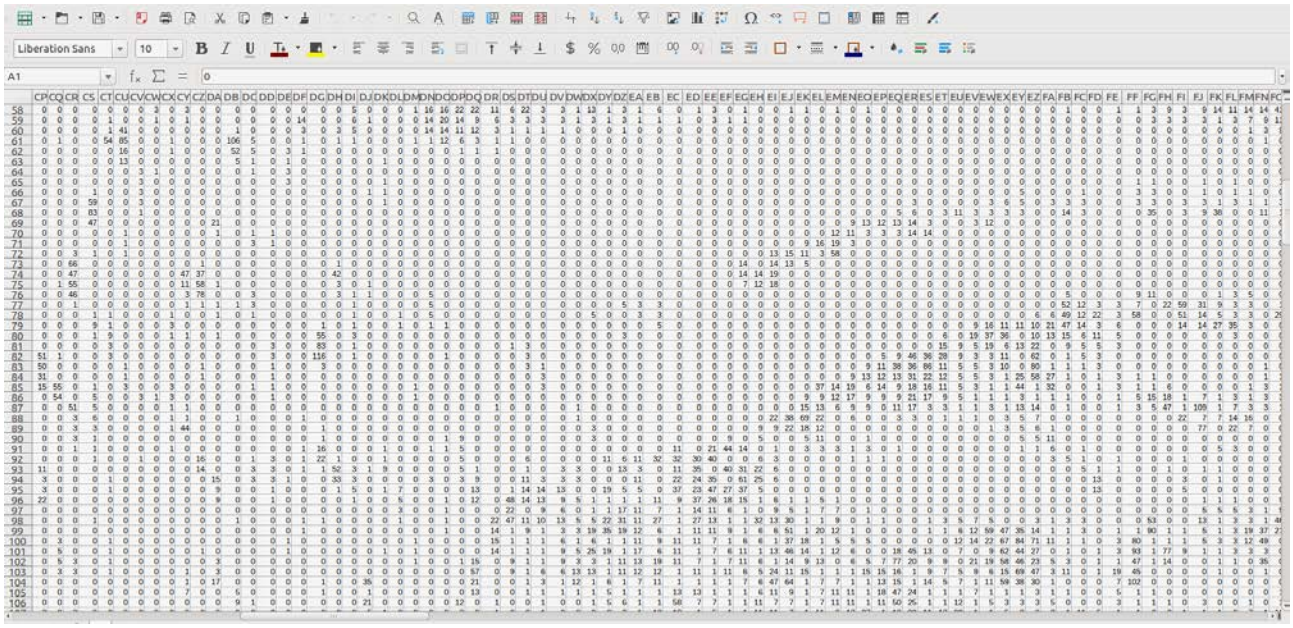


fig 39: Iteración en la que se descarta cada pixel de una sección de una imagen cerca de un ojo

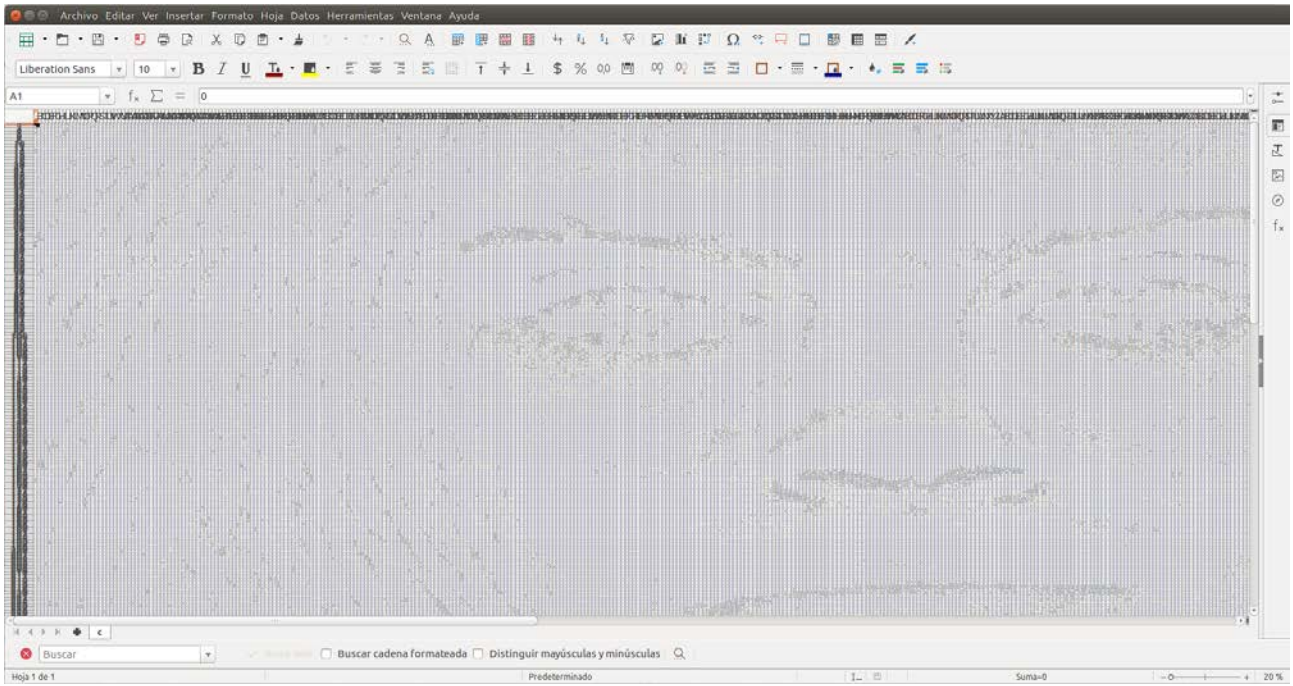


fig 40: Ventana de un procesador de hojas de calculo visionando un CSV generado en este experimento con un zoom-out alto

Analizando estas imágenes podemos ver que el porcentaje de celdas que se descartan en la primera iteración es de entre el 70% y el 75% en imágenes con caras y de más del 80% en imágenes sin caras.

La iteración media de descarte esta en las fotografías con cara entre la 3ª y la 4ª iteración y en aquellas que no presentan caras en torno a la 2ª iteración.

Podemos apreciar que existe una ventaja aparente en la condición de descarte inherente al algoritmo de detección que estamos utilizando.

5.2. Verificación de la plataforma base

Para la verificación del hardware inicial se desarrolla un IP core sencillo que simplemente realiza una tramsferencia de video entre la interfaz AXIS de lectura y la interfaz AXIS de escritura.

Introducimos una señal de video HDMI generada mediante una SBC Raspberry pi que nos permite ofrecer una imagen fija o mostrar la señal capturada a través de una cámara conectada en un terminal USB o CSI. La salida HDMI se conectará a un monitor. La plataforma de verificación que se utiliza sigue el siguiente esquema:

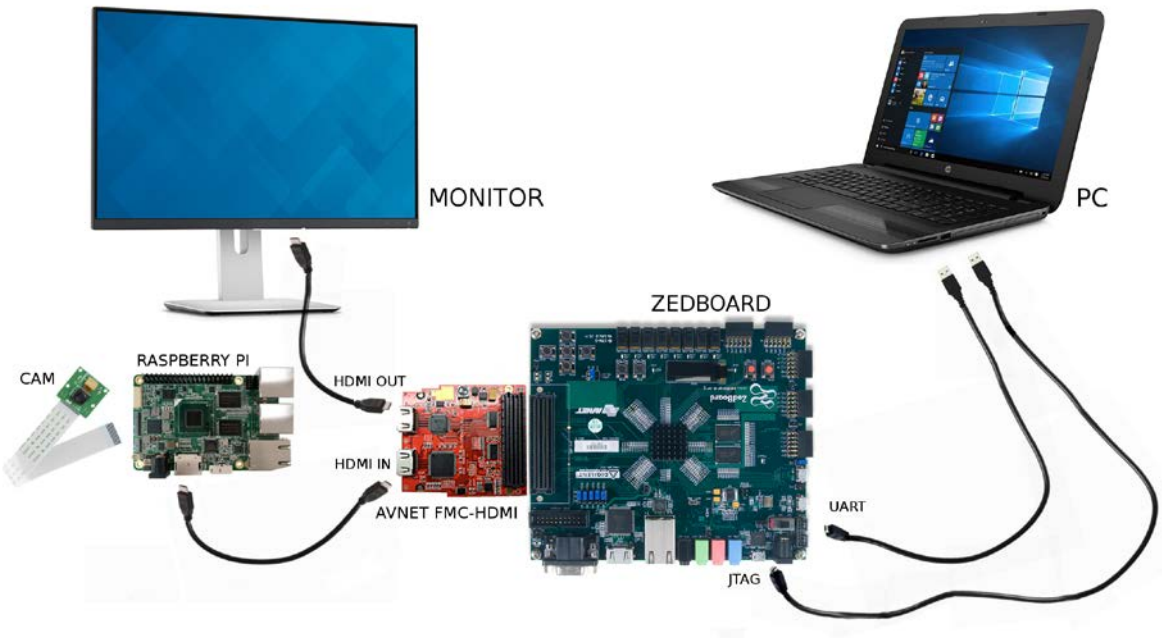


fig 41: Esquema de la plataforma de desarrollo y verificación

En el PC tendremos acceso a la consola de la aplicación a través de un terminal UART y a la interfaz JTAG para poder programar y depurar la placa.

El JTAG nos permitirá depurar el sistema en dos niveles: podremos depurar el software utilizando el depurador GDB de la plataforma SDK y podremos depurar el hardware utilizando los cores de analizador lógico ILA disponibles en la plataforma VIVADO.

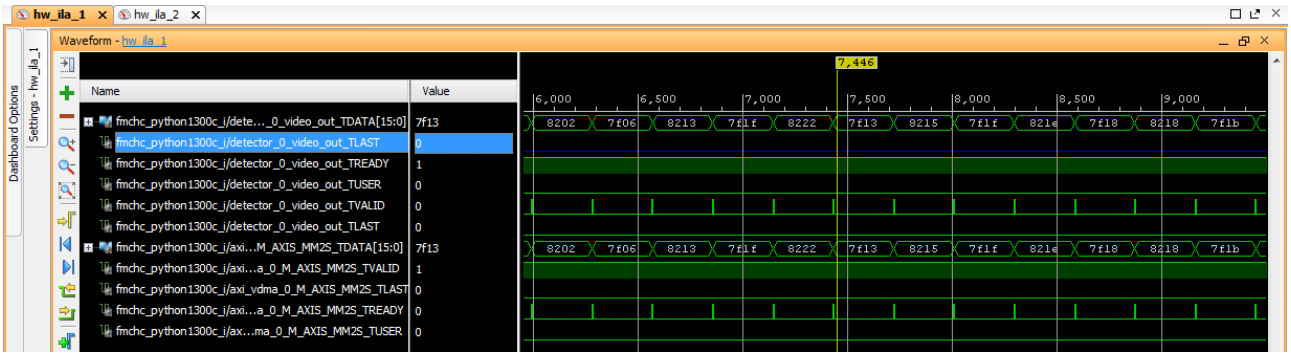


fig 42: Ejemplo de captura de datos realizada con ILA

5.3. Verificación del sistema en placa

El proceso de verificación del sistema a diseñar es complejo. Desde que hacemos una modificación en el código C hasta que el resultado es comprobado en la plataforma FPGA final se han de llevar a cabo las siguientes etapas:

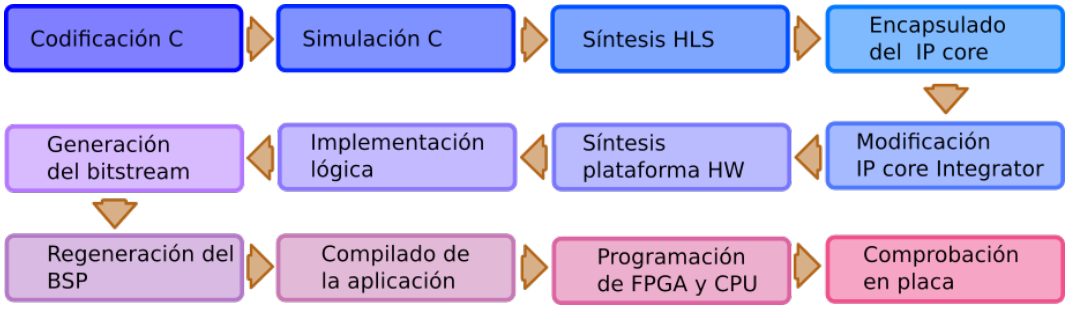


fig 43: diagrama de flujo de desarrollo en HLS

El proceso completo puede tardar decenas de minutos o incluso horas dependiendo de la complejidad de las síntesis. Por lo tanto es muy importante ser metódico en la verificación de cada etapa.

Los requisitos que debe cumplir nuestro diseño se concentran sobre todo en el tiempo de respuesta. El objetivo en cuanto a tiempo de respuesta es, en principio, el suficiente como para garantizar la ejecución de la aplicación en tiempo real. Los requisitos de tiempo real dependen de las necesidades de la aplicación final. Así por ejemplo, la persistencia de la imagen en el ojo es de aproximadamente 1/16 de segundo, por lo que para una señal de video podríamos hablar de tiempo real a partir de esa tasa de 16 frames por segundo. Si la aplicación final del detector es acompañar un eye-tracking o cualquier otra aplicación de interfaz hombre-máquina, la tasa podría ser incluso un poco menor. Si el detector de objetos debe procesar cada uno de los frames de entrada, la tasa de transferencia más habitual para video es de 25 fps pero puede ser de incluso 60fps o más en algunos casos y no se consideraría tiempo real si se perdieran frames.

Si marcamos como referencia esos 16 fps podemos establecer el tiempo del que disponemos para procesar cada pixel si queremos garantizar esos requisistos. Así, para una secuencia de video de resolución VGA en un segundo moveríamos

$$640px * 480px * 16fr = 4.915.200pixels$$

Teniendo en cuenta que sincronizamos nuestro circuito con un reloj de 150Mhz, disponemos de 30 ciclos de reloj para procesar cada uno de los pixels.

Si la resolución fuera de 1080p, el número de pixels por segundo sería:

$$1920px * 1080px * 16fr = 33.177.600pixels$$

y en este caso deberíamos procesar cada pixel en 4 ciclos y medio.

La herramienta HLS Vivado nos ofrece, después de realizar una síntesis, una serie de reportes con estimaciones sobre las necesidades de recursos y sobre el tiempo de ejecución.

Ambos parámetros pueden variar después de la síntesis e implementación final del circuito, pero la diferencia suele ser mínima.

El reporte de tiempos nos da información, además de sobre si se estima que la lógica podrá trabajar a la frecuencia definida, sobre la latencia y el tiempo de reentrada de los diferentes segmentos (loops, funciones, etc) en que se divide el programa.

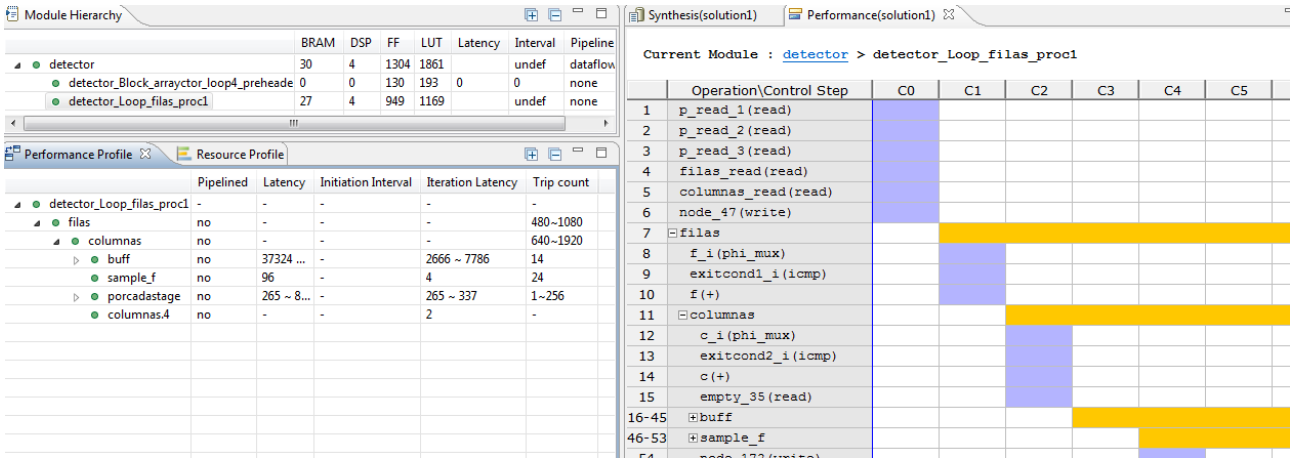


fig 44: Ejemplo de un reporte después de una síntesis HLS

Estos reportes son, después de la simulación funcional en C, el segundo paso de verificación de nuestro sistema. Analizando estos datos con minuciosidad podemos saber, por un lado si se cumplen las especificaciones y por otro donde está el cuello de botella o que parte del programa está produciendo los retrasos.

Este tipo de análisis permite anticiparse y obtener una medición sobre los resultados en pocos minutos sin la necesidad de llegar a realizar el ensayo en la placa, lo cual supone un tiempo mucho mayor. Sin embargo, como alcanzar los objetivos no es sencillo y conlleva días e incluso semanas de trabajo, limitarse a esta fuente de verificación puede ser contraproducente teniendo en cuenta que la estrategia o la línea que se lleva a cabo puede provocar fallos en etapas posteriores. Si esto ocurriese podríamos haber perdido mucho tiempo si el fallo implica cambios drásticos que perjudican de nuevo la consecución de los requisitos de síntesis HLS.

Para el desarrollo de este proyecto se ha diseñado y trabajado siguiendo el siguiente esquema de verificación para tratar de optimizar el tiempo de desarrollo y acelerar el proceso.

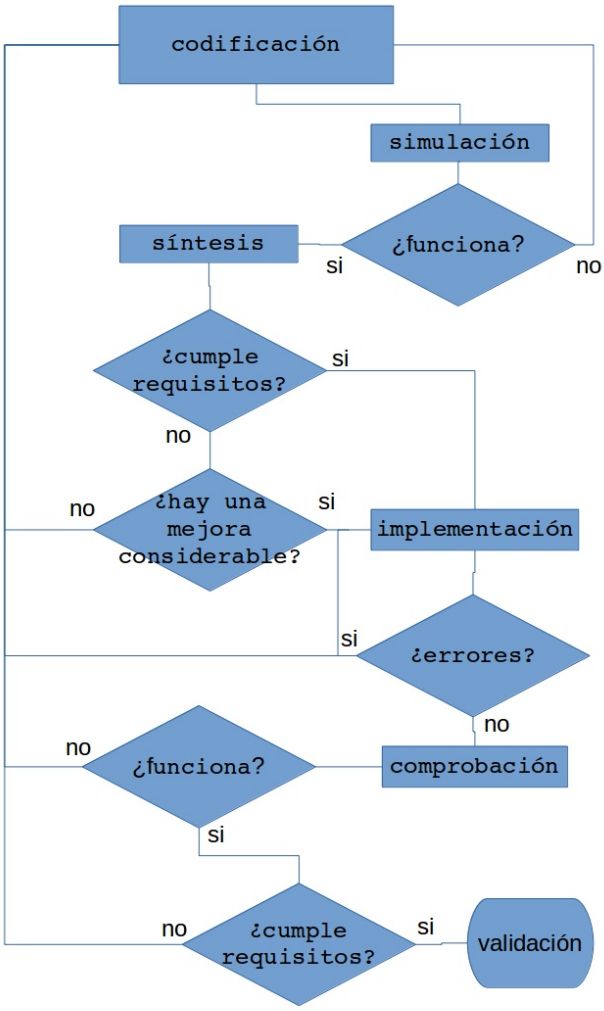


fig 45: diagrama de flujo del proceso de verificación del programa en HLS

5.4. Verificación de la aplicación final

Para verificar el sistema hardware/software una vez implementado en la placa de desarrollo necesitamos comprobar, por un lado el correcto funcionamiento y por otro que se cumplen las especificaciones temporales.

Para comprobar el funcionamiento utilizamos las mismas fotografías usadas para comprobar el algoritmo y que fueron descritas en la sección “Verificación funcional del algoritmo”. Se cargarán las imágenes con una escala especificada a través de la interfaz

HDMI de la Raspberry Pi utilizando alguna aplicación de visionado de imágenes y se compararán los resultados con los obtenidos en el PC.

Después se comprobará con las mismas imágenes la velocidad de procesado. Para ello utilizaremos el contador de frames que incluimos en la implementación y que leemos a través de la variable de salia dummy.

La tasa de frames por segundo se calcula como:

$$\frac{(2ª LecturadeDummy) - (1ª LecturadeDummy)}{Tiempoentreambaslecturas}$$

A través de la aplicación podemos programar un temporizador para aguardar un intervalo pequeño de segundos y comprobar el número de frames procesados durante ese tiempo. Este proceso lo repetiremos con distintas imágenes de entrada y para diferentes resoluciones.

6. Resultados

Se han medido los resultados del desarrollo atendiendo a diferentes parámetros. A continuación se muestran y valoran para cada una de las secciones del desarrollo dichos parámetros.

6.1. Resultados del diseño del algoritmo en PC

No se ha hecho un estudio demasiado exhaustivo sobre el funcionamiento del detector ya que se parte de un clasificador entrenado con antelación. Se ha comprobado que la codificación del algoritmo fuese correcta y se ha testado con una muestra de 44 imágenes tal y como se describe en el apartado de “verificación”

El resultado es que de todas las fotos que hemos seleccionado de una manera más o menos aleatoria sobre caras frontales en un buscador WEB, solamente una falla a la hora de localizar sus ojos.

De cara a tener una referencia para determinar la rapidez del detector que implementemos en hardware hemos medido el tiempo de respuesta de procesado de un conjunto de imágenes. Los datos han sido generados con un programa que realiza el test sobre las imágenes aplicando varias escalas. El programa genera un documento CSV que puede ser analizado con una hoja de cálculo. A continuación se muestra uno de los documentos generados donde se han evaluado 17 imágenes de resolución VGA

imagen	columnas	filas	t a esc 4x(5/7)	hits	t a esc 3x(5/7)	hits	t a esc 2x(5/7)	hits	t a esc 5/7	hits	t	hits	HITS
c1.bmp	640	480	5.158	10	6.177	20	12.339	15	22.15	1	40.239	1	47
c2.bmp	640	480	4.628	1	6.962	5	13.11	5	26.231	3	52.1	2	16
c3.bmp	640	480	6.169	2	6.846	4	13.029	0	23.54	0	43.574	0	6
c4.bmp	640	480	6.742	14	7.434	1	13.736	3	23.205	0	40.826	0	18
c5.bmp	640	480	5.418	1	6.392	3	13.37	1	25.864	1	48.114	0	6
c6.bmp	640	480	5.16	7	8.035	0	16.7	0	32.163	1	57.875	0	8
c7.bmp	640	480	2.829	0	5.038	0	12.385	0	21.289	0	41.624	0	0
c8.bmp	640	480	4.67	0	5.771	0	11.672	0	21.555	0	41.219	0	0
c9.bmp	640	480	2.416	0	4.043	6	8.825	11	16.519	0	31.06	0	17
c10.bmp	640	480	4.01	7	5.643	17	11.554	12	20.263	0	37.417	0	36
c11.bmp	640	480	3.353	1	5.963	11	12.511	1	24.275	0	45.146	0	13
c12.bmp	640	480	4.196	0	6.004	0	12.453	0	25.045	0	49.699	0	0
c13.bmp	640	480	6.011	12	9.399	13	19.282	1	30.555	3	54.682	1	30
c14.bmp	640	480	3.428	3	4.772	12	10.318	20	18.047	14	34.828	0	49
c15.bmp	640	480	4.777	0	7.269	2	15.481	32	20.155	50	37.892	50	134
c16.bmp	640	480	3.865	0	10.741	0	14.688	6	29.856	17	56.452	24	47
c17.bmp	640	480	4.499	1	5.149	0	9.633	3	18.599	36	36.308	32	72

fig 46: Tiempo de respuesta y número de objetos localizados en una muestra de 17 imágenes sobre un Intel® Core™ i5-5200U CPU @ 2.20GHz × 4 - 8MB RAM

Calculando la media obtenemos un valor de algo más de 22 frames/segundo para una resolución VGA sin escalado. Vemos como el número de elementos detectados no supone una diferencia significativa con muestras donde no se han producido detecciones.

La imagen c17 con 32 detecciones tiene un tiempo de respuesta menor que la mayoría de las imágenes que no han encontrado objetos.

6.2. Resultados de la plataforma hardware base

Se ha validado que la plataforma base funciona correctamente. Se ha comprobado que:

a) Es capaz de recibir e identificar señales de video de entrada de diferentes resoluciones hasta un máximo de 1920*1080 establecido en el diseño.

b) La plataforma muestra las señales escritas en el Frame Buffer de salida siempre con una resolución de 1920*1080. Podría ajustarse la resolución de salida pero el driver no está desarrollado. Si la imagen que se almacena tiene una resolución menor, solo ocupará la parte proporcional de la pantalla completandose el resto con un fondo verde.

c) La interfaz de usuario funciona correctamente y podemos reiniciar el sistema utilizando el comando ‘start’.

d) Al plataforma acepta adecuadamente los bloques desarrollados en HLS con las interfaces especificadas: entrada AXIS de video, salida AXIS de video, bus AXI4-Lite; tal y como se describe en las especificaciones de diseño.

e) El pixel rate entre los Frame Buffers es de 150000000pixeles/segundo. Por lo tanto, es capaz de disponer/recibir un pixel por ciclo de reloj.

f) El número de recursos utilizados en el XC7Z020 (sin añadir ningún IP core de procesado) se muestra en la siguiente imagen (fig 48):

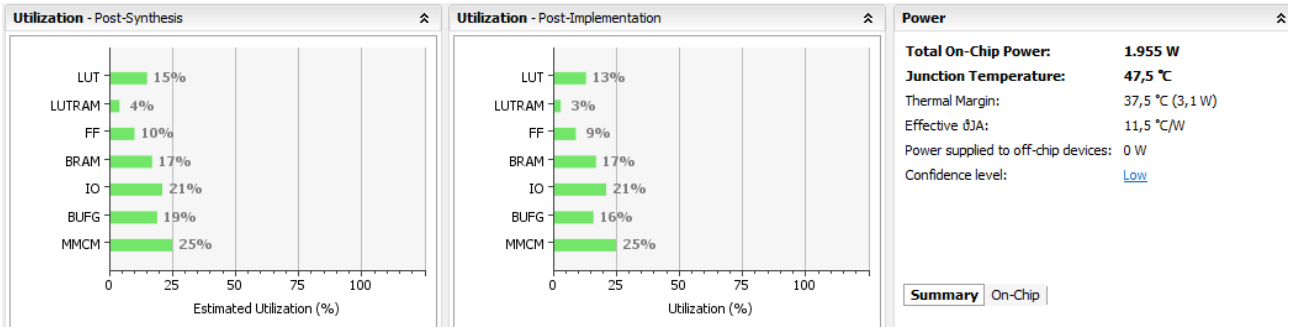


fig 47: Resultados de utilización y potencia estimada tras la síntesis e implementación de la plataforma base

El porcentaje de BRAM utilizados podría ajustarse reduciendo el tamaño de los buffers de algunos IP cores.

6.3. Resultados del IP ‘detector’

Los resultados de rendimiento del detector implementado utilizando síntesis de alto nivel no han sido tan buenos como se pretendía.

La última versión del IP desarrollada antes de redactar estas líneas conseguía procesar un frame VGA en un tiempo de 570ms. El tiempo de respuesta en este caso es

fijo, ya que no se aplica la condición de descarte. Se procesan los 256 stages para cada pixel independientemente del resultado de las hipótesis.

Synthesis(solution1)

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	6.00	5.14	0.75

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
85708909	597196909	85708910	597196910	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	954
FIFO	-	-	-	-
Instance	2	4	898	13654
Memory	25	-	71	42
Multiplexer	-	-	-	1209
Register	-	-	4540	44
Total	27	4	5509	15903
Available	280	220	106400	53200
Utilization (%)	9	1	5	29

Detail

fig 48: Resultados de la síntesis HLS del detector

Performance Profile | Resource Profile

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
detector	-	85708909~597196909	85708910 ~ 597196910	-	-
ini_hits	no	25	-	1	25
filas_columnas	no	85708800 ~ 597196800	-	288	297600~2073600
porcadastage	yes	269	1	15	256
resultado	no	75	-	3	25

fig 49: Análisis de tiempo para el IP core sintetizado

Los valores de latencia que pueden observarse en las tablas corresponden a una resolución VGA (85708909 ciclos de reloj [570ms]) y 1080p (597196909 ciclos de reloj [3981ms]).

El resultado en frames por segundos es de 1.75. Este dato está comprobado sobre la plataforma hardware base.

La síntesis que se ofrece no se ha realizado haciendo ningún tipo de paralelización en las dimensiones fila o columna ya que todos los intentos hasta el momento han sido fallidos.

El valor de frame-rate más alto que podamos conseguir sin paralelizar a una frecuencia de 150Mhz ocurriría teóricamente en el caso de poder procesar cada pixel en un ciclo de reloj. En este caso, para una resolución VGA, obtendríamos 488 frames por segundo.

Con las condiciones de la implementación que hemos desarrollado, en la cual procesamos todos los “stages” a pesar de que ya se haya producido la condición de descarte (algún clasificador no haya superado su umbral) y donde además, paralelizamos utilizando una estrategia de tipo pipeline, la frecuencia más alta desde el punto de vista teórico que podemos alcanzar sería de 1.9 frames por segundo que es el valor frontera al que nos acercamos. Esto es debido a que, bajo estas premisas, necesitaríamos, al menos, 256 ciclos de reloj (uno por cada stage del detector) para procesar un pixel.

Si pudiésemos implementar la condición de descarte sobre el bucle en pipeline con la misma eficiencia (1 ciclo de througput) y teniendo en cuenta el valor empírico obtenido en la etapa de verificación en cuanto al valor medio de stages analizados antes del descarte en las imágenes (entre 3 y 4 iteraciones de media por pixel), podríamos acercarnos teóricamente a los 120 frames por segundo.

Sin embargo, utilizando la condición de descarte no hemos conseguido sintetizar ningún programa con menos de 13 ciclos de diferencia entre iteraciones en el pipeline (througput). El resultado de estas síntesis probadas sobre la plataforma hardware base es muy similar en tiempo al de aquellas síntesis que no introducían la condición de descarte.

7. Conclusiones

El desarrollo de este proyecto ha sido sin duda un trabajo apasionante. Con un objetivo preciso y bien definido desde el principio, el proceso ha sido una escalada constante para acercarnos paso a paso al final.

En cuanto a las herramientas seleccionadas, hemos podido comprobar que realmente las nuevas plataformas y metodologías de desarrollo en el campo de la lógica programable permiten un desarrollo acelerado trabajando con un alto nivel de abstracción. Un trabajo de esta envergadura no hubiera cabido quizá en el ámbito de un proyecto de fin de carrera hace unos pocos años si no contáramos con los SoC de la familia Zynq o con la plataforma de desarrollo Vivado.

Sin embargo, aunque el desarrollo se acelere, la curva de aprendizaje de la herramienta es aun bastante tendida. Una vez alcanzado el Know-how necesario, el trabajo de desarrollo es mucho más liviano, pero las herramientas son aun complejas y poco intuitivas y dejan entrever con frecuencia la complejidad del proceso que envuelven.

Para la consecución del proyecto ha sido necesario afrontar, con más o menos intensidad, un conjunto muy amplio de las fases de desarrollo de sistemas hardware. Desde la evaluación de plataformas de desarrollo, el diseño y preparación de una plataforma base hasta la implementación de la aplicación final.

Hemos comprobado que la optimización de sistemas es la parte más complicada y que este es un trabajo lento que requiere mucha creatividad y perseverancia además de un gran dominio del sistema y de las herramientas. Es en este punto donde los autores encuentran la duda de si el trabajo a alto nivel con síntesis HLS es realmente más eficiente y rápido que el trabajo a bajo nivel con HDL. En el momento en el que el requerimiento de optimización es moderadamente estricto, queremos decir.

Para finalizar, aunque los resultados no hayan sido del todo satisfactorios en cuanto a la implementación obtenida, sí que lo han sido en el nivel pedagógico y personal. Por otro lado consideramos que el trabajo, aunque sí concluido, no está terminado ni cerrado. Estamos seguros de que con el tiempo suficiente será posible alcanzar e incluso mejorar los objetivos planteados ya que quedan aún sendas por recorrer en cuanto a estrategias de mejora de la síntesis HLS. Además, en cuanto a que el producto que desarrollamos no

es una aplicación final, sino que es una herramienta con mucha proyección se abre un gran abanico de proyectos y trabajos futuros.

7.1. Trabajo futuro

Como trabajo futuro más inmediato proponemos tratar de mejorar los tiempos de respuesta del programa HLS para poder alcanzar, al menos, una tasa igual a la del stream de entrada de video. La propuesta de mejora consiste en buscar nuevas estrategias de paralelización como la adhesión de canales o el uso de DSPs para hacer operaciones combinadas.

Por otro lado también se propone migrar y probar el IP core en otras plataformas como la Zynqberry donde se puede utilizar como dispositivo de entrada una cámara raspberry pi Cam.

En cuanto a la proyección del detector una vez concluido y empaquetado, existen un sin fin de posibles aplicaciones.

Una propuesta que tiene un especial sentido para los autores de este trabajo es el desarrollo de una interfaz hombre-máquina a través de los ojos. Existen muchas propuestas parecidas, muchas de ellas basadas en la técnica del Eye-Tracking que consiste en que, a través de una cámara podamos saber dónde estamos mirando y utilizar esta información para controlar un apuntador como el cursor del ratón. Los sistemas de Eye-tracking actuales se basan casi todos en la grabación de uno o dos ojos utilizando una cámara fija a una distancia constante del ojo.

Si utilizamos el sistema que hemos desarrollado para buscar ojos en imágenes con mayor resolución, podemos conseguir que la interfaz esté libre del aparatoso hardware que hasta el momento se utiliza. En este sentido, el Eye-Tracking se realizaría sobre la sección de la imagen donde se han detectado ojos. De esta forma podríamos utilizar estas interfaces que pueden facilitar la accesibilidad a personas con discapacidad en servicios públicos como cajeros automáticos, oficinas de información, dispensadores, etc simplemente con una cámara fija. Quizá sería necesario añadir en paralelo un detector de ojos cerrados para poder simular eventos como el click del ratón. Una interfaz de alta resolución como ratones basados en Eye-Tracking obtiene un nivel alto de eficiencia con cámaras VGA. Para una aplicación de baja resolución como una interfaz de cajero automático sería suficiente con 16 a 32 posiciones que requerirían una resolución de la imagen que contiene al ojo de entre 48 y 96 pixels cuadrados.

Una cámara con una apertura de visión de 60º y resolución VGA podría hacerlo a una distancia de entre 40 y 80 cm. Una cámara 1080p podría conseguir esta resolución a más de 2 metros. El módulo raspberry cam v2 tiene una resolución de 8Mp (2464p).

Otro campo donde se está investigando mucho en el ámbito de la detección de objetos (y no precisamente ojos) es el del aforamiento, control y medida de velocidad de coches en tránsito.

8. Referencias

[1] P. Viola and Jones: "*Rapid Object Detection using a Boosted Cascade of Simple Features", 2001 http://wearables.cc.gatech.edu/paper_of_week/viola01rapid.pdf*

[2] Yoav Freund and Robert E. Schapire: *"A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting",* Journal of Computer and System Sciences, 1997 *http://www.face-rec.org/algorithms/Boosting-Ensemble/decision-theoretic_generalization.pdf*

[3] *Zedboard de Avnet web page: http://zedboard.org/product/zedboard*

[4] AES-FMC-HDMI-CAM-G de Avnet web page : *https://www.avnet.com/shop/us/p/kits-and-tools/development-kits/avnet-engineering-services/aes-fmc-hdmi-cam-g-3074457345628965509*

[5] VIVADO HLx Design Suite de Xilinx: *https://www.xilinx.com/products/design-tools/vivado.html* , Xilinx web page.

[6] Xilinx Software Development Kit (XSDK) de Xilinx, Xilinx web page: *https://www.xilinx.com/products/design-tools/embedded-software/sdk.html*

[7] Xilinx: "AXI Reference Guide", UG761, 2011 *https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf*

[8] Raúl Rojas: *"AdaBoost and the Super Bowl of Classifiers. A Tutorial Introduction to Adaptive Boosting"* Computer Science Departament Freie Universität Berlín, 2009 *http://www.inf.fu-berlin.de/inst/ag-ki/adaboost4.pdf*

[9] Yoav Freund and Robert E. Schapire: "A Short Introduction to Boosting", Journal of Japanese Society for Artificial Intelligence, 1999 *https://cseweb.ucsd.edu/~yfreund/papers/IntroToBoosting.pdf*

[10] Hradiš, M.; Herout, A. & Zemčík, "*Local Rank Patterns - Novel Features for Rapid Object Detection*" Proceedings of International Conference on Computer Vision and Graphics, 2008

[11] Adam Herout, Pavel Zemčík, Michal Hradiš, Roman Juránek, Jiří Havel, Radovan Jošth and Lukáš Polok: *"Low-Level Image Features for Real-Time Object Detection", cap6. Adam Herout, ISBN 978-953-7619-90-9, 2010*

[12] Roman Juránek, Pavel Zemčík, Adam Herout "*Implementing the Local Binary Patterns with SIMD Instructions of CPU*" WSCG Poster papers, 2010.

[13] Pavel Zemčík, Roman Juránek, Petr Musil, Martin Musil, Michal Hradiš: "*HIigh Performance Architecture For Object Detection In Streamed Videos*", 978-1-4799-0004-6/13, 2013

[14] Jan Sochman, Jirí Matas: "WaldBoost – Learning for Time Constrained Sequential Detection" CVPR, 2005

[15] Zemcik, P., Hradis, M., Herout, A.: "*Local Rank Differences - Novel Features for Image Processing"*, Proceedings of SCCG 2007

[16] Yi-Qing Wang: "An Analysis of the Viola-Jones Face Detection Algorithm" Image Processing On Line, 2014 *http://dx.doi.org/10.5201/ipol.2014.104*

[17] Yu Wei, Xiong Bing, and C. Chareonsak. "*Fpga implementation of adaboost algorithm for detection of face biometrics."* Biomedical Circuits and Systems, 2004

[18] T. Theocharides, N. Vijaykrishnan, and M.J. Irwin. "*A parallel architecture for hardware face detection*". Emerging VLSI Technologies and Architectures, 2006.

[19] Lukáš Polok, Adam Herout, Pavel Zemčík, Michal Hradiš, Roman Juránek, and Radovan Jošth. "*-local rank differences- image feature implemented on gpu*", 2008

[20] Xilinx All Programable, "Vivado Design Suite User Guide. High-Level Synthesis" UG902, 2017.
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug902-vivado-high-level-synthesis.pdf

[21] Xilinx All Programable, "*Vivado Design Suite User Guide. Designing IP Subsystems. Using IP Integrator*" UG994, 2017
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug994-vivado-ip-subsystems.pdf

[22] Xilinx All Programable, "*UltraFast High-Level Productivity Design Methodology Guide*" UG1197, 2017 *https://www.xilinx.com/support/documentation/sw_manuals/ug1197-vivado-high-level-productivity.pdf*

[23] Pedram Azad, Tilo Gockel, Rüdiger Dillmann, "*Computer Vision, Principles and Practice*", Elektor, ISBN 978-0-905705-71-2, 2008

[24] OpenCV web page: *http://opencv.org/*

[25] Avnet Electronic Marketing documentation. “FMC-HDMI-CAM+PYTHON-1300-C Frame Buffer Design Tutorial”, 2016

[26] Stephen Neuendorffer, Thomas Li, and Devin Wang: “*Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries*”, Xilinx XAPP1167, 2015.

[27] Xilinx All Programable, “Video In to AXI4-Stream V4.0” LogiCORE IP Product Guide, PG043, 2015

[28] Xilinx All Programable, “AXI Video Direct Memory Access V6.2” LogiCORE IP Product Guide, PG020, 2016

[29] Xilinx All Programable, “Clocking Wizard V5.3” LogiCORE IP Product Guide, PG065, 2016

[30] Xilinx All Programable, “Video Timing Controller V6.0” LogiCORE IP Product Guide, PG016, 2013

[31] Xilinx All Programable, “AXI4-Stream to Video Out V4.0” LogiCORE IP Product Guide, PG044, 2015

[32] Xilinx All Programable, “AXI IIC Bus Interface V2.0” LogiCORE IP Product Guide, PG090, 2016

[33] Xilinx All Programable, “AXI Interconnect V2.1” LogiCORE IP Product Guide, PG059, 2017

[34] Xilinx All Programable, “Processor System Reset Module V5.0” LogiCORE IP Product Guide, PG164, 2015

[35] Xilinx All Programable, “Processing System 7 V5.5” LogiCORE IP Product Guide, PG082, 2017

[36] Avnet Electronic Marketing documentation. “*HDMI Input/Output FMC Module with Camera Interface. Hardware Guide*”, 2015

[37] Analog Devices Datasheet: “ADV7511 Low-Power HDMI Transmitter with Audio Return Channel”, 2011

[38] Analog Devices Datasheet: “ADV7611 Reference Manual”, UG180, 2017

[39] ANSI/VITA 57.1-2008 (R2010), FPGA Mezzanine Card (FMC) Standard