Trabajo Fin de Máster
"Máster Universitario en Microelectrónica: Diseño y Aplicaciones de Sistemas Micro/Nanométricos"

Diseño de sistemas empotrados para aplicaciones de procesado de imagen y vídeo sobre FPGAs usando Vivado SDSoC

Autor / Author Roberto Joaquín Del Pino Roldán

Tutor / Advisor María José Avedillo de Juan

Santiago Sánchez Solano

Fecha / Date Septiembre 2019

Santiago Sánchez Solano

María José Avedillo de Juan

Agradecimientos

Mi más sincero agradecimiento a María José Avedillo de Juan y Santiago Sánchez Solano, por su orientación, interés y apoyo en cada una de las etapas.

A mi pareja, mi familia y mis amigos por ayudarme a seguir adelante.

Resumen

El procesado de imagen y vídeo es un campo que tiene una amplia área de aplicaciones, abarcando desde la automatización del hogar hasta servicios de seguridad y aplicaciones militares. Por otra parte, las aplicaciones emergentes en entornos como IoT demandan el desarrollo de sistemas empotrados con elevada potencia de cálculo pero recursos de tamaño, coste y consumo de potencia reducido.

Una solución eficiente para suplir dicha demanda es el uso de compuertas programables (FPGA). Los dispositivos Zynq All Programmable System on Chip (AP SoC) de Xilinx tienen la capacidad de procesar grandes cantidades de datos, ya que incorporan un sistema de procesamiento (PS) de altas prestaciones junto a lógica programable (PL).

Las capacidades de los dispositivos Zynq de Xilinx, junto con las nuevas metodologías y herramientas de desarrollo que permiten la descripción de algoritmos a alto nivel y facilitan la exploración del espacio de diseño con el objetivo de alcanzar compromisos adecuados coste/rendimiento, resultan particularmente útiles en la implementación de algoritmos de procesado de imágenes y vídeo, en los que se explota el paralelismo inherente de los algoritmos de procesado a través de la asignación de tareas de cómputo intensivo al hardware.

El presente Trabajo Fin de Máster expone el desarrollo de un sistema básico de procesado de imagen y vídeo, implementado sobre la placa de desarrollo ZYBO (Zynq BOard) que incorpora un dispositivo AP SoC de la familia Zynq de Xilinx. El desarrollo del sistema está basado en una metodología de codiseño hardware/software con la herramienta SDSoC.

La metodología de desarrollo incluye el diseño de una plataforma hardware para el entorno SDSoC, la implementación y optimización de algoritmos capaces de operar en tiempo real y la verificación de estos sobre la placa de desarrollo.

El sistema de detección de bordes diseñado recibe un flujo de vídeo de entrada en formato 1080p@60Hz a través del puerto HDMI presente en la placa de desarrollo. Este flujo de entrada es procesado mediante un algoritmo, acelerado en lógica programable, y se encuentra disponible a la salida del sistema a través del puerto VGA de la placa.

La funcionalidad del sistema se demuestra mediante la implementación de algoritmos de procesado (detección de bordes Sobel, filtro laplaciano y filtro sepia), en los que se puede apreciar de forma clara el procesado en tiempo real que realizan los algoritmos acelerados en hardware frente a la pérdida de frames de vídeo de sus semejantes implementados en el PS, es decir, sin aceleración.

Palabras clave: FPGA, Zynq - 7000, SDSoC, HLS, frame-buffer, procesado de imagen y vídeo, aceleración software, Xilinx.

Índice general

1. Introducción y objetivos

1.1. Justificación

La visión es posiblemente el sentido humano más importante, razón por la cual el procesado de imagen y video juega un papel muy importante en nuestra sociedad, siendo un amplio campo de investigación en constante evolución. El interés en el procesado de imagen y vídeo se centra principalmente en grandes grupos de tareas como son la mejora y extracción de la información pictográfica, o el procesamiento de los datos para almacenamiento, transmisión y representación.

El primer procesado de imágenes se registró a partir de 1957, cuando se agregó un escáner a una computadora en la Oficina Nacional de Estándares en los Estados Unidos. Fue utilizado para algunas de las primeras investigaciones sobre detección de bordes y reconocimiento de patrones. En la década de 1960, la necesidad de procesar grandes cantidades de imágenes de gran tamaño, obtenidas de satélites y exploración espacial, estimuló la investigación del procesamiento de imágenes en el Laboratorio de Propulsión a Chorro de la NASA.

A medida que las computadoras incrementaron su potencia y se redujo su coste, hubo una explosión en el rango de aplicaciones para el procesado de imagen y vídeo, abarcando desde imágenes médicas hasta inspección industrial. En la Tabla 1 se muestran algunas de las aplicaciones y el ámbito de uso de sistemas de procesado de imagen y vídeo.

Tabla 1. *Aplicaciones y ámbito de uso del procesado de imagen y vídeo.*

Ámbito se uso	Aplicaciones
Medicina	• Rayos X • Resonancia Magnética

Automoción	• Sistemas Avanzados de Asistencia al Conductor: – Reconocimiento de señales de tráfico. – Sistemas de alerta de salida de carril. – Estacionamiento asistido.
Identificación biométrica	• Reconocimiento facial. • Reconocimiento de huellas dactilares. • Reconocimiento mediante análisis de retina.
Agricultura	• Monitoreo y seguimiento del ganado. • Monitoreo de los sistemas de riego y el estado de las cosechas.
Industria	• Análisis de piezas en líneas de producción. • Clasificación de piezas. • Inspección automática del producto final.
Aeroespacial	• Geoprocesamiento. • Análisis espectral.
Militar	• Reconocimiento de objetivos. • Vehículos de combate autónomos.

las aplicaciones emergentes en entornos como IoT demandan el desarrollo de sistemas empotrados con elevada potencia de cálculo pero recursos de tamaño, coste y consumo de potencia reducido. Una solución eficiente para suplir dicha demanda es el uso de compuertas programables (FPGA).

Las capacidades de procesamiento de los dispositivos Zynq All Programmable System on Chip (AP SoC) de Xilinx son particularmente útiles en la implementación de sistemas de procesado de imágenes y vídeo. El procesamiento de grandes cantidades de datos de píxeles se puede llevar a cabo mediante su estructura FPGA de lógica programable (PL), mientras que en su sistema de procesamiento (PS) se pueden implementar los algoritmos más complejos. Además, los dispositivos Zynq cuentan con herramientas que facilitan el diseño de este tipo de aplicaciones, entre las que podemos destacar las siguientes:

- Xilinx IP blocks — Existen varios bloques de propiedad intelectual (IP) para aplicaciones de procesado de imágenes y video disponibles en la herramienta IP Integrator, incluyendo memoria de video, mejora de imagen y funcionalidad de ajuste de color entre otros.

- Vivado HLS Video Libraries — Vivado HLS incluye soporte específico para el procesado de imagen y video, a través de una biblioteca de funciones sintetizables en lenguaje de descripción hardware (HDL).

Aún más, con el fin de facilitar el co-diseño hardware/software y aprovechar las ventajas de los dispositivos Zynq, Xilinx ha introducido SDSoC™, un novedoso entorno de desarrollo integrado (IDE) que facilita la síntesis de alto nivel a partir de descripciones C/C++. Este entorno permite a los desarrolladores software aprovechar toda la potencia hardware y software de los dispositivos, eliminando así el reto que supone el diseño a nivel de transferencia de registros (RTL) en cuanto a la experiencia que debe poseer el diseñador para sacar partido a los recursos hardware disponibles.

1.2. Objetivos

El objetivo principal de este Trabajo Fin de Máster es la exploración y la documentación de una metodología de diseño de sistemas de procesado de imagen y video en tiempo real sobre APSoCs de Xilinx. En concreto, se utiliza la placa de desarrollo ZYBO que contiene un dispositivo ZYNQ XC7Z010-1CLG400C y el entorno de diseño SDSoC. Los objetivos concretos son:

1. Desarrollar una plataforma hardware base con entrada HDMI y salida VGA sobre la que se puedan implementar distintos algoritmos de procesado. Documentar este desarrollo para que pueda replicarse para diferentes placas/dispositivos o versiones del entorno de diseño.
2. Desarrollar aceleradores hardware para distintos algoritmos de procesado usando la herramienta de síntesis de alto nivel HLS integrada en SDSoC. Extraer directrices genéricas para la optimización de aceleradores hardware.
3. Diseñar, validar y evaluar sistemas de procesado de video con aceleración hardware. Documentar el procedimiento para facilitar su uso a otros miembros del equipo de trabajo que deben abordar el desarrollo de este tipo de aplicaciones.

1.3. Estructura

La Memoria se ha estructurado como sigue:

En el Capítulo 2 se introducen los conocimientos y conceptos previos con los que ha sido necesario familiarizarse para el desarrollo de este trabajo. También se incluye una breve revisión de las distintas herramientas de Xilinx que soportan las diferentes metodologías o flujos de diseño que pueden emplearse para el desarrollo de este tipo de sistemas.

En el Capítulo 3 se describe la plataforma hardware base desarrollada utilizando el procesador ARM Cortex-A9 incluido en el dispositivo ZYNQ-7000 y un conjunto de módulos IP proporcionados por el fabricante.

En el Capítulo 4 se describen los elementos básicos y el flujo de diseño asociados al entorno SDSoC y se detalla el desarrollo de un sistema que incorpora en la cadena de procesado de video una etapa de detección de bordes implementada en hardware. Se detalla el diseño y la optimización de dicho acelerador. El Capítulo concluye con un apartado en el que se reportan y evalúan otros algoritmos de procesado de video desarrollados con la metodología explorada. De nuevo la descripción detallada de los procedimientos se ha incluido como anexos que puedan utilizarse como manuales/tutoriales.

Finalmente se exponen las conclusiones de este trabajo y se esbozan líneas de trabajo futuro.

2. Fundamentos

La realización de este trabajo requiere conocimientos previos que se resumen en este Capítulo y que se clasifican en tres áreas diferentes. En primer lugar se introducen conceptos básicos del procesado de imagen y video, campo de aplicación en el que se va a trabajar. En segundo lugar se describe la palca de desarrollo sobre la que se implementan nuestras soluciones. Finalmente, se introducen las herramientas de diseño que se han utilizado.

2.1. El procesado de imagen y vídeo

Desde un punto de vista teórico, una imagen se define como una función de dos dimensiones f(x,y), donde 'x' e 'y' son las coordenadas de un plano que contiene todos los puntos de la misma y f(x,y) es la amplitud en el punto (x,y), denominada intensidad o nivel de gris de la imagen.

Cuando las coordenadas (x,y) y los valores de intensidad son discretos y finitos, se habla de una imagen digital. Cada uno de los elementos de una imagen digital tiene una localización y un valor particular denominados puntos elementales de la imagen o píxeles, siendo este último término comúnmente utilizado para denotar la unidad mínima de medida de una imagen digital.

El fundamento de una imagen digital en color es el mismo, salvo que cada elemento o píxel es descrito y codificado según el espacio de color que se utilice. Estos espacios de color se describirán posteriormente.

Por otro lado, un vídeo digital es una representación de imágenes en movimiento, cuyo contenido evoluciona con el tiempo. De forma general, un vídeo es una función de tres dimensiones, dos en el espacio y una en el tiempo.

En el estudio clásico del procesamiento de señales digitales, las señales suelen ser funciones del tiempo de una dimensión, sin embargo, las imágenes digitales y el video son señales multidimensionales. Las imágenes son funciones de dos y quizás tres dimensiones espaciales, mientras que el video digital como una función también incluye una tercera (o cuarta) dimensión de tiempo. La Figura 2.1 muestra una representación de la dimensionalidad de las imágenes y el vídeo.

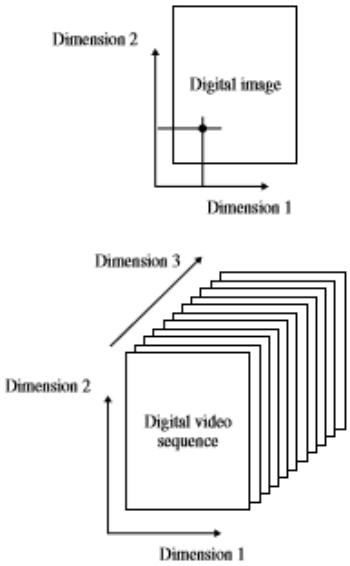


Figura 2.1. *Dimensionalidad de imágenes y vídeo.*

En el campo del procesado de imagen y vídeo, el procesado de imágenes se refiere a imágenes simples o fijas, mientras que el procesado de video se refiere a una serie temporal de imágenes (frames), generalmente a una velocidad de frame específica. Existen numerosas técnicas para el procesado de imagen y video, que pueden clasificarse de forma general como sigue:

- Mejorar o alterar las imágenes.
- Extraer información de ellas.
- Comprimir datos de imagen y video.

El estándar de video de Alta Definición (full HD) utiliza un tamaño de imagen de 1920 x 1080 píxeles, equivalente a 2.073.600 píxeles en total. Teniendo en cuenta estas grandes dimensiones, cualquier procesamiento que implique píxeles individuales requiere que se realicen un gran número de cálculos en paralelo, una tarea que es ideal para una FPGA. Por otro lado, los algoritmos más complejos que operan con menos datos pueden implementarse en el software.

Muchas aplicaciones requieren operación en "tiempo real". Los sistemas que soportan dichas aplicaciones deben procesar los datos lo suficientemente rápido para que la información sea procesada al mismo ritmo que se va generando. Debido al gran volumen de datos involucrado, el diseño de algoritmos rápidos mediante el uso de aceleradores hardware es una prioridad importante.

2.1.1. Espacios de color

El color posee unas características innatas que permiten distinguir un color de otro. Estas características son:

- ***Tono*:** es el valor cromático de un color, la frecuencia del espectro donde se encuentra. Es la cualidad que permite clasificar los colores por su nombre (amarillo, rojo, verde, etc.)

- ***Luminosidad*:** es el resultado de la mezcla de los colores con blanco o negro y tiene referencia de matiz. Representa la cantidad de luz presente en un color, más blanco o más negro, según sea el caso. Cuanto mayor es la luminosidad, mayor es la cantidad de luz, es decir, más color blanco posee.

- ***Saturación*:** se refiere al grado de pureza de un color y se mide con relación al gris. Los colores con menor saturación se muestran más agrisados, con mayor cantidad de impurezas y menor intensidad luminosa.

Los espacios de color permiten describir el color y descomponerlo en distintos canales. En esencia, un espacio de color es la especificación de un sistema de coordenadas tridimensional y de un subespacio de este sistema, en el que cada color queda representado por un único punto. Entre los espacios de color más comunes en el procesado de imágenes y vídeo se encuentra el modelo RGB, HSI, YUV, YCbCr y CMY.

Modelo RGB

Es un modelo aditivo de mezcla, basado en 3 colores primarios: Rojo, Verde y Azul. En este modelo las intensidades de los colores primarios se suman para producir otros colores. Está basado en el sistema de coordenadas cartesianas y su representación gráfica es en un cubo unitario como el que se muestra en la Figura 2.2.

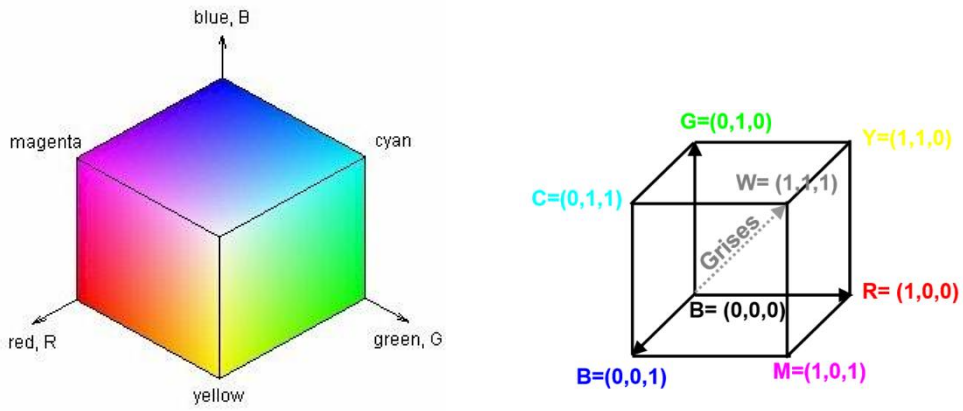


Figura 2.2. *Espacio RGB*

El modelo RGB es uno de los más utilizados para crear y reproducir los colores en monitores y pantallas. Por otro lado, muchas de las cámaras digitales utilizan el formato RGB, lo que hace que este modelo sea relevante en el procesado de imágenes y vídeo.

Modelo HSI

Este modelo caracteriza el color en términos de tono (**H**ue), saturación (**S**aturation) e intensidad (**I**ntensity). Se define a través de una transformación no lineal del espacio de color RGB, modificando el subespacio del cubo de la Figura 2.2 y convirtiéndolo en dos conos unidos por la base, tal como se muestra en la Figura 2.3. Las transformaciones matemáticas que permiten el paso del espacio RGB al HSI se realizan normalmente mediante hardware específico, debido a su alto coste computacional.

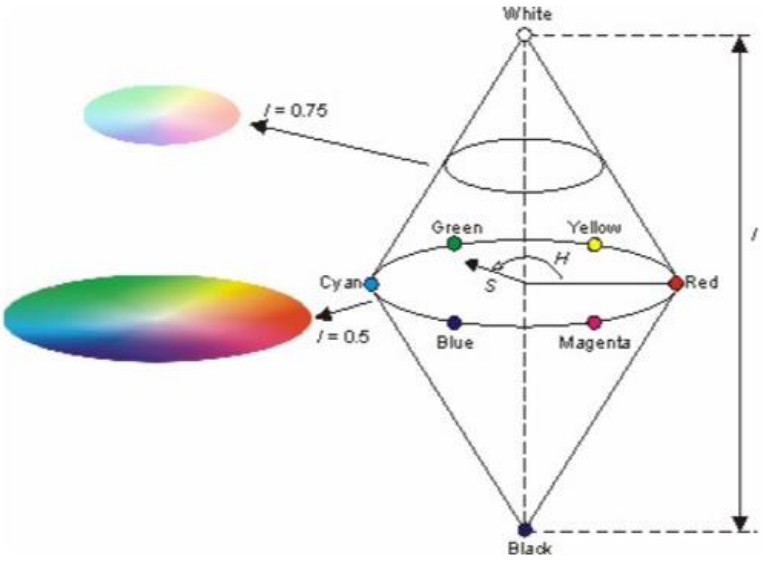


Figura 2.3. *Espacio HSI*

El componente de saturación (componente S) se corresponde con la distancia radial de dicho cono, proporcionando una medida del grado en el que un color está mezclado con la luz blanca. Por otra parte, el tono (componente H) corresponde al ángulo respecto al eje rojo, proporcionando una magnitud de la longitud de onda dominante. Por ejemplo, como se puede observar en la Figura 2.3, cuando H = 0º el color representado es el rojo, mientras que cuando H = 60º, el color que se representa es el amarillo.

La componente de intensidad (componente I) se obtiene como la distancia a lo largo del eje perpendicular al plano del color, indicando el valor del brillo. Valores bajos de I corresponden a colores oscuros, mientras que valores superiores corresponden a colores claros hasta llegar al blanco.

Las características de este modelo tienen similitud con la forma en que los humanos percibimos el color, convirtiéndolo en una herramienta ideal para desarrollar algoritmos de procesado de imágenes y vídeo. No obstante, la complejidad computacional que presenta hace que su implementación sea poco práctica.

Modelo YUV

Es el formato de color utilizado por los estándares NTSC, PAL y SECAM. El modelo YUV define un espacio de color en términos de una componente de luminancia (brillo) y dos componentes de crominancia (color). Y es el componente de luminancia, mientras que U y V marcan la crominancia.

Está más próximo al modelo humano de percepción del color que el estándar RGB, aunque no tan cerca como el HSI. Normalmente U y V son submuestreados por un factor de dos a cuatro en la dimensión espacial, ya que el ojo humano es mucho menos sensible a estos factores. Esto es muy útil, por ejemplo, para reducir el número de bits utilizado en técnicas de compresión de imágenes.

La conversión del modelo YUV a RGB se puede expresar mediante las siguientes ecuaciones:

$Y = 0.299R + 0.587G + 0.114B$

$U = -0.147R - 0.289G + 0.436B$

$V = 0.615R - 0.515G - 0.100B$

Modelo YCbCr

El modelo YCbCr (también conocido como Y'CbCr) es una codificación no lineal del espacio RGB. El espacio YCbCr es una versión escalada y desplazada del espacio de color YUV. El parámetro Y indica la luminancia, los parámetros Cb y Cr indican el tono del color: Cb ubica el color en una escala entre el azul y el amarillo, Cr indica la ubicación del color entre el rojo y el verde.

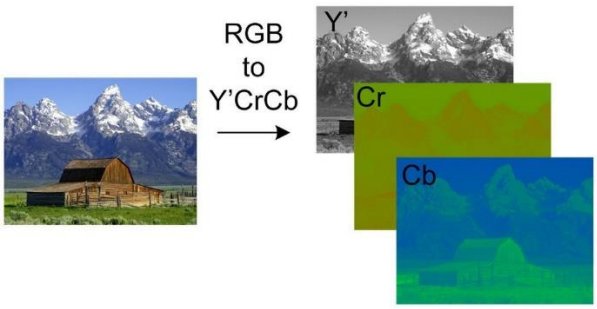


Figura 2.4. *Representación de una conversión RGB -> YCbCr.*

En la práctica, generalmente se usa en los estudios de televisión europeos y en la compresión de imágenes.

Modelo CMY

Es un modelo de color definido con los colores primarios cian, magenta y amarillo (CMY). Es útil para describir la salida de color de los dispositivos de copia (p. ej. impresoras). A diferencia de los monitores de video, que producen un modelo de color aditivo, en este modelo los colores se obtienen por luz reflejada, lo cual es un proceso sustractivo. Los dispositivos hardware como las impresoras producen una imagen de color cubriendo un papel con pigmentos de color.

La mayoría de los dispositivos que depositan pigmentos coloreados sobre papel, tales como impresoras y fotocopiadoras en color, necesitan una entrada CMY o bien una conversión interna de RGB a CMY.

Dado que cian, magenta y amarillo son los colores secundarios, para convertir los colores primarios en secundarios se ha de realizar la siguiente transformación:

$$\begin{pmatrix} C \\ M \\ Y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

El sistema de coordenadas es el mismo que en el modelo RGB pero donde había negro ahora existe blanco y viceversa

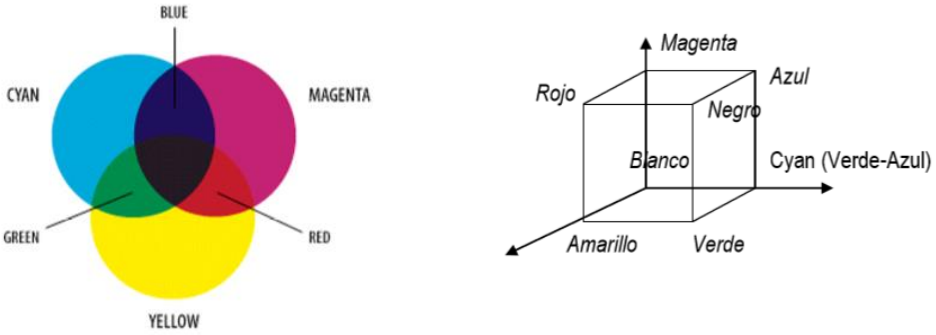


Figura 2.5. *Espacio CMY.*

2.1.2. Niveles de abstracción

El procesado de imágenes y vídeo comprende distintos tipos de procesamiento sobre diferentes tipos y volúmenes de datos. Por ello, es usual dividirlo en tres niveles de abstracción, que se caracterizan por el volumen de datos que se procesan y la cantidad de conocimiento disponible sobre el contenido de la imagen. En la Figura 2.6 se puede observar una representación de estos niveles de abstracción.

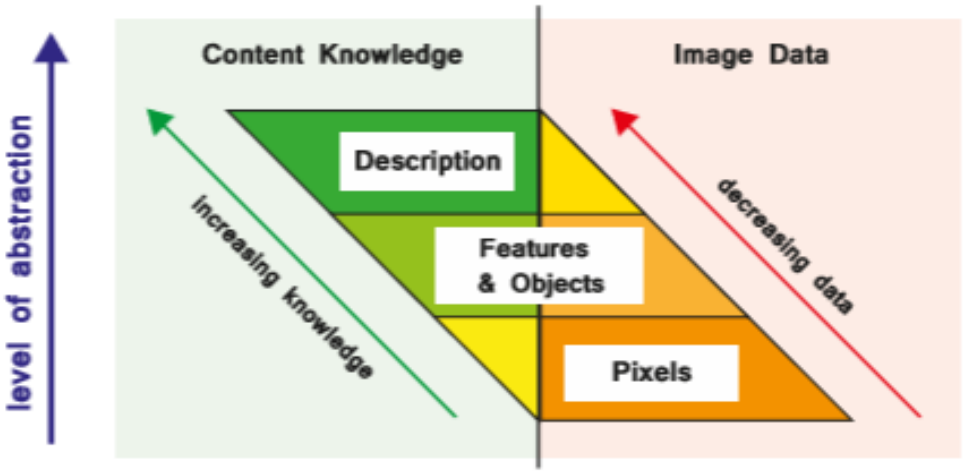


Figura 2.6. *Niveles de abstracción en el procesado de imagen y vídeo.*

Como se comentó anteriormente, un vídeo es una representación de imágenes que evolucionan con el tiempo. En lo que sigue, se definirán los niveles de abstracción haciendo referencia a imágenes, entendiéndose que esto incluye también el flujo de vídeo.

Píxeles

El procesado a nivel de píxeles representa el nivel más bajo de abstracción, con la mayor cantidad de datos para procesar y muy poco conocimiento sobre el contenido y el significado de la imagen. En este nivel se incluyen, entre otros, los ajustes en el balance de color o el contraste de una imagen, o los algoritmos en el dominio espacial para el filtrado mediante píxeles vecinos (por ejemplo las operaciones de suavizado para reducir el ruido o el filtrado Sobel para resaltar los bordes [1], [2], [3]).

Por lo general, estas tareas a nivel de píxel, en las que puede haber varias etapas, preceden a otras operaciones y se suelen denominar "procesamiento previo".

Características y objetos

Como parte del proceso de extracción de información de una imagen, se detectan las características y los objetos. Esto incluye la identificación de líneas, curvas, formas o regiones. Se pueden emplear varias técnicas para ayudar a lograr la transición de píxeles a características y objetos, como pueden ser la transformada de Hough, la identificación del color, el umbral o la morfología [1], [2], [3].

Descripción

Una vez identificadas las características u objetos relevantes en la imagen, la etapa final es lograr el conocimiento de lo que representa la imagen, es decir, una descripción de su contenido. En esta etapa, la información no es una imagen, sino una descripción textual o numérica. Lograr una descripción implica extraer parámetros de las características identificadas e interpretarlas o clasificarlas según una comparación con criterios predeterminados [1], [2], [3].

2.1.3. Señales de sincronización

Independientemente del nivel de abstracción, el primer problema en la creación de un canal de procesamiento de video (pipeline) es qué hacer con las señales de sincronización. En un nivel fundamental, todos los frames de video tienen tres regiones:

- Período de refresco vertical (vertical blanking).
- Período de refresco horizontal (horizontal blanking).
- Región activa.

La región activa es la ubicación del frame donde residen los píxeles de la imagen, en la cual trabaja el algoritmo. También son los datos a los que se hace referencia

implícitamente cuando un diseñador comienza a pensar en el procesado de video. Las regiones horizontal y vertical proporcionan el espaciado de sincronización requerido para que otros equipos de video detecten y muestren el contenido correctamente. Para determinar la viabilidad en un sistema de tiempo real en el que la pérdida de un frame puede no ser aceptable, debemos calcular el tiempo entre dos frames de video consecutivos y establecer la frecuencia de reloj requerida.

Un período de frame de video consiste en intervalos de tiempo activos y no activos, también llamados intervalos de borrado. Durante los intervalos activos, los datos de píxeles de video se transfieren. Históricamente, los monitores CRT (tubo de rayos catódicos) utilizaron intervalos de refresco horizontal y vertical para reposicionar el haz de electrones desde el final de una línea hasta el comienzo de la siguiente línea (horizontal blanking) o desde el final de un frame hasta el inicio de otro (vertical blanking). La Figura 2.7 ilustra los períodos de refresco y activos de un frame de video.

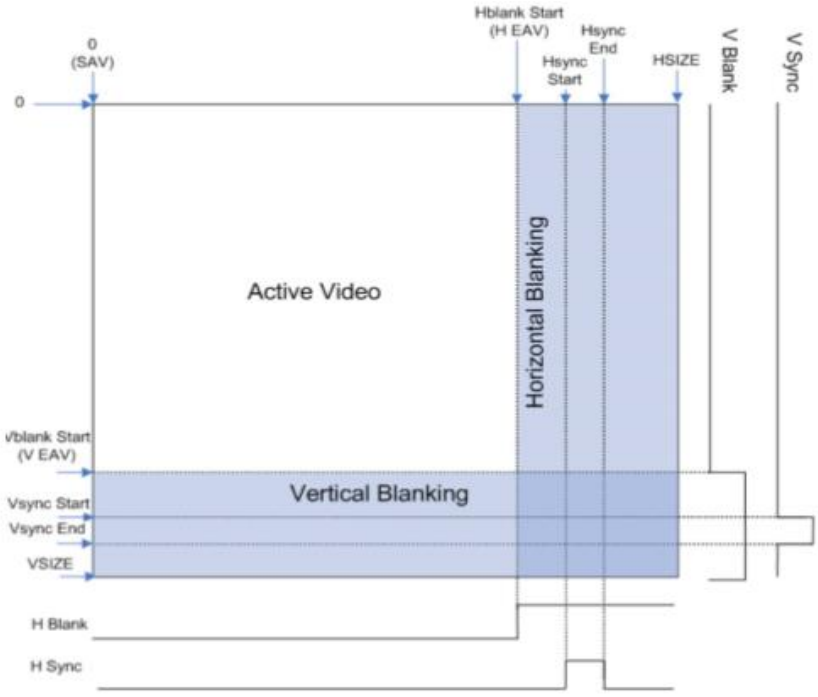


Figura 2.7. *Temporización en un frame de vídeo [4].*

En una resolución de vídeo de 1920 x 1080 píxeles con una tasa de refresco de 60 frames o fotogramas por segundo (1080p60), el período de refresco vertical representa 45 píxeles por línea y el período de refresco horizontal de 280 líneas por fotograma de video. La frecuencia de reloj de video requerida se calcula como (1920 + 280) x (1080 + 45) x 60 Hz = 148.5 MHz.

Se puede consultar [5] para obtener detalles sobre los parámetros de tiempo de video.

2.2. Placa de desarrollo ZYBO

El trabajo se ha realizado sobre la placa de desarrollo de sistemas empotrados ZYBO (diminutivo de ZYnq BOard) de Digilent.

ZYBO es una plataforma de desarrollo diseñada alrededor del miembro más pequeño de la familia Xilinx Zynq-7000, el Z-7010 [6], que Integra un procesador ARM Cortex-A9 de doble núcleo con la estructura lógica de una FPGA de la serie Xilinx 7. El dispositivo Zynq Z-7010 puede albergar el diseño de un sistema completo cuando se combina con el amplio conjunto de periféricos multimedia y de conectividad disponibles en ZYBO. La Figura 2.8 muestra cómo ZYBO incluye memoria integrada, entrada / salida de video y audio, Ethernet, varios GPIO, seis conectores Pmod y más en una tarjeta compacta.

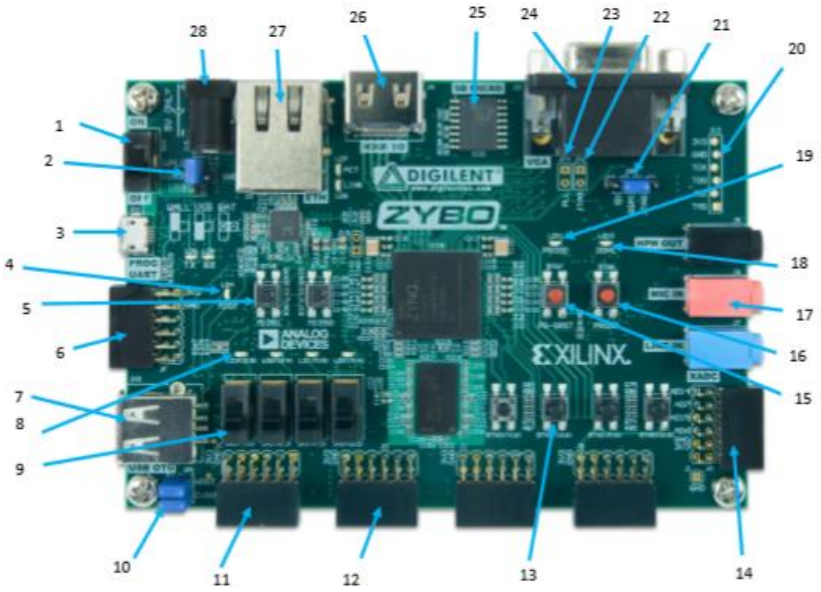


Figura 2.8. *Plataforma de desarrollo ZYBO [6].*

Cada uno de los componentes numerados en la Figura 2.8 se describe en la Tabla 2.

Tabla 2*. Componentes de la placa ZYBO [5].*

Callout	Component Description
1	Power Switch
2	Power Select Jumper and battery header
3	Shared UART/JTAG USB port

Callout	Component Description
15	Processor Reset Pushbutton
16	Logic configuration reset Pushbutton
17	Audio Codec Connectors

4	MIO LED	18	Logic Configuratio Done LED
5	MIO Pushbuttons (2)	19	Board Power Good LED
6	MIO Pmod	20	JTAG Port for optional external cable
7	USB OTG Connectors	21	Programming Mode Jumper
8	Logic LEDs (4)	22	Independent JTAG Mode Enable Jumper
9	Logic Slide switches (4)	23	PLL Bypass Jumper
10	USB OTG Host/Device Select Jumpers	24	VGA connector
11	Standard Pmod	25	microSD connector (Reverse side)
12	High-speed Pmods (3)	26	HDMI Sink/Source Connector
13	Logic Pushbuttons (4)	27	Ethernet RJ45 Connector
14	XADC Pmod	28	Power Jack

Esta plataforma de desarrollo integrado proporciona un reloj de 50 MHz (Zynq PS_CLK). La entrada de 50 MHz permite que el procesador funcione a una frecuencia máxima de 650 MHz y que el controlador de memoria DDR3 funcione a un máximo de 525 MHz (1050 Mbps). El diseño del sistema base de ZYBO

A continuación se describen los tres componentes fundamentales que se utilizan en nuestro sistema: el dispositivo Zynq, en el cual se implementa el procesado, y los componentes HDMI y VGA, que se utilizan como interfaces de entrada y salida de vídeo del sistema.

2.2.1. Dispositivos Zynq-7000

La arquitectura general Zynq se muestra en la Figura 2.9. Consta de dos partes, el sistema de procesamiento (PS) y la lógica programable (PL), que pueden usarse de forma independiente o en conjunto. De hecho, los circuitos de alimentación están configurados con dominios separados para cada sección, permitiendo que la parte PS o PL se apague si no está en uso.

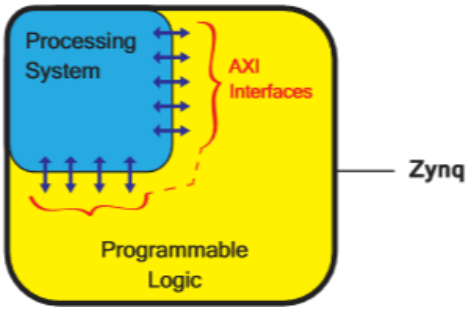


Figura 2.9. *Modelo simplificado de la arquitectura Zynq [7].*

La sección PL es ideal para implementar lógica de alta velocidad, aritmética y subsistemas de flujo de datos, mientras que la sección PS admite rutinas y/o sistemas operativos, lo que significa que la funcionalidad general de cualquier sistema diseñado puede dividirse adecuadamente entre el hardware y el software. Las conexiones entre la lógica programable y el sistema de procesamiento se llevan a cabo mediante el estándar "Interfaz Avanzada Extensible (AXI)" [8], que describiremos posteriormente.

Sistema de procesamiento (PS)

Todos los dispositivos Zynq tienen como base del sistema de procesamiento un procesador ARM Cortex-A9 de doble núcleo. Este es un procesador "*hard*", es decir, existe como un elemento de silicio dedicado y optimizado en el dispositivo.

Con fines de comparación, la alternativa a un procesador *hard* es un procesador "*soft*" como Xilinx MicroBlaze, que puede formarse combinando elementos de la estructura de lógica programable [9], como se muestra en la Figura 2.10. La implementación de un procesador *soft* es, por lo tanto, equivalente a cualquier otro bloque IP implementado en la estructura lógica de un FPGA. En general, la ventaja de los procesadores *soft* es que el número y la implementación de las instancias del procesador son flexibles. Por otro lado, los procesadores *hard* pueden lograr un rendimiento considerablemente mayor, como ocurre con el procesador ARM de Zynq.

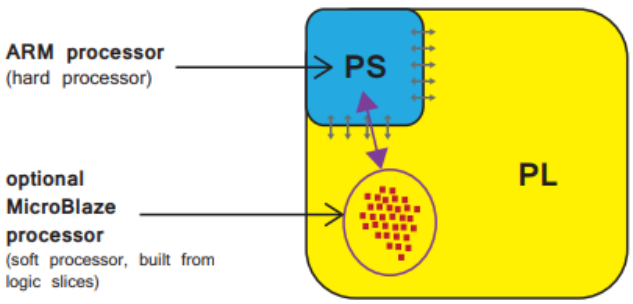


Figura 2.10. *Localización del procesador "soft" (Microblaze) en Zynq [7].*

Es importante destacar que el sistema de procesamiento Zynq comprende no solo el procesador ARM, sino un conjunto de recursos asociados que forman una Unidad de procesamiento de aplicaciones (APU); y otras interfaces periféricas, memoria caché, interfaces de memoria, interconexión y circuitos de generación de reloj. Un diagrama de bloques de la arquitectura PS se muestra en la Figura 2.11.

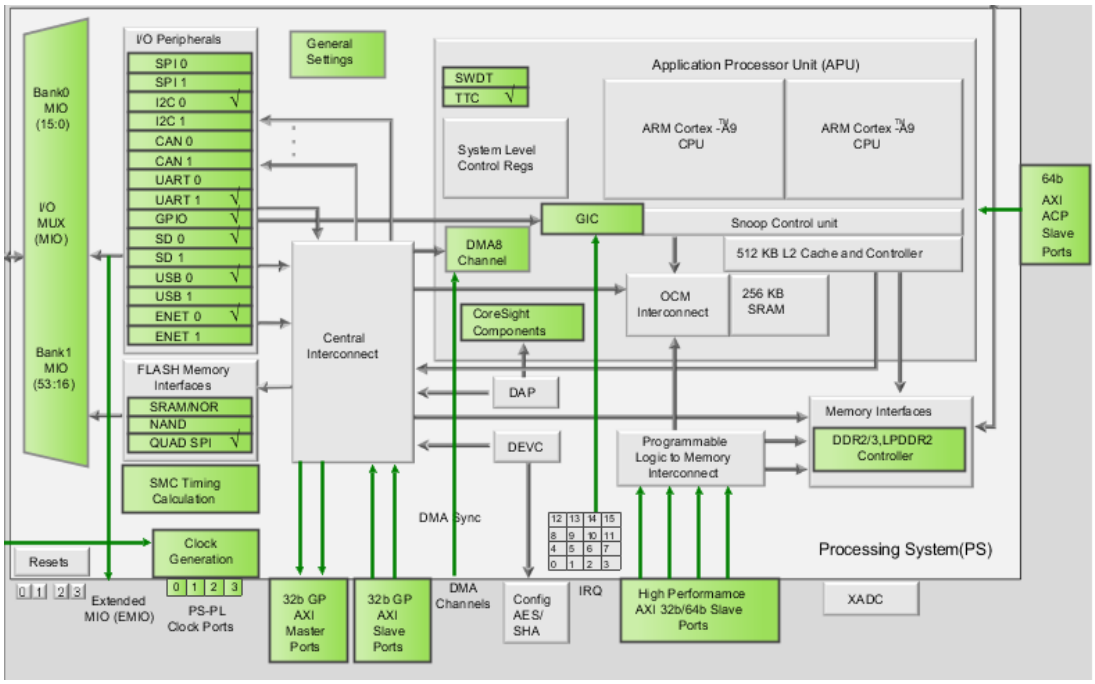


Figura 2.11. *Diagrama de bloques del Sistema de Procesamiento (PS) [10].*

La APU se compone principalmente de dos núcleos de procesamiento ARM, cada uno con unidades computacionales asociadas: una Unidad de Punto Flotante (FPU), una Unidad de Gestión de la Memoria (MMU) y una memoria caché de nivel 1 (en dos secciones para instrucciones y datos).

Esta unidad también contiene una memoria caché de nivel 2 y una memoria adicional en chip (OCM). Finalmente, una unidad de control Snoop (SCU) forma un puente entre los núcleos ARM y la memoria caché de nivel 2 y las memorias OCM. La SCU además gestiona las transacciones que tienen lugar entre el PS y el PL a través del puerto de coherencia del acelerador (ACP), que se describe posteriormente. En la Figura 2.12 la se muestra un diagrama de bloques simplificado de la APU

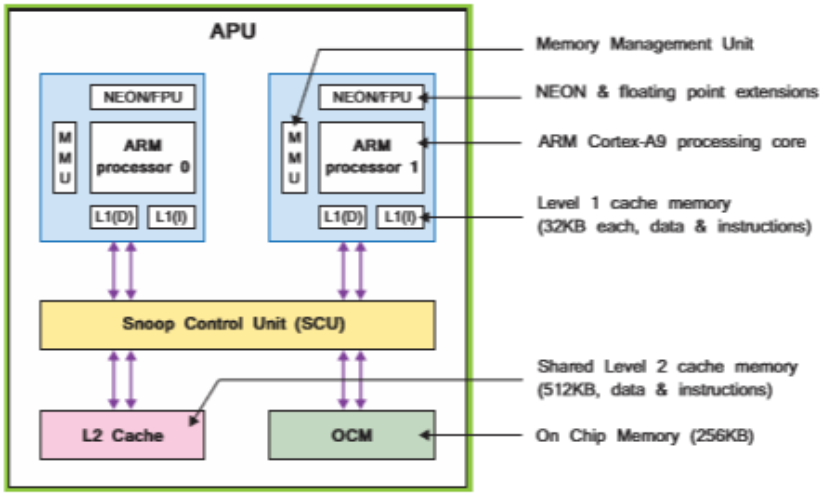


Figura 2.12. *Diagrama de bloques simplificado de la APU [11].*

Los controladores de periféricos del PS contienen registros de control direccionables en el espacio de memoria de los procesadores y están conectados a estos. Esta comunicación entre el PS y las interfaces externas se logra principalmente a través de la entrada / salida (E/S) multiplexada (MIO), que proporciona 54 pines de conectividad flexible. Los controladores de periféricos que no tienen sus entradas y salidas conectadas a los pines MIO pueden enrutarlas mediante la interfaz MIO extendida (EMIO) que se encuentra dentro de la lógica programable (PL).

Los puertos de E/S disponibles están compuestos por interfaces de comunicaciones estándar; y entrada / salida de uso general (GPIO) que se puede utilizar para una variedad de propósitos, incluyendo botones simples, interruptores y LED. En el Manual técnico de referencia de Zynq7000 [11] se encuentra disponible información detallada sobre cada una de estas interfaces.

Lógica Programable (PL)

La estructura lógica programable (PL) de un dispositivo Zynq se representa en la Figura 2.13. Está compuesta principalmente por una estructura lógica FPGA de propósito general. Los diseños pueden implementar múltiples bloques en la estructura FPGA, cada uno de los cuales contiene registros de control direccionables. Además, los módulos implementados en PL pueden activar interrupciones en los procesadores y realizar accesos directos (DMA) a la memoria DDR3.

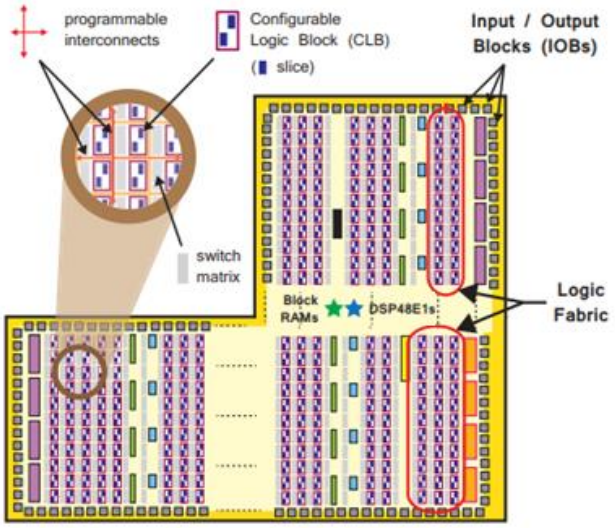


Figura 2.13. *Estructura Lógica Programable de un dispositivo Zynq [7].*

Las principales características de la estructura PL pueden resumirse en los siguientes elementos:

- ***Bloque Lógico Configurable (CLB):*** los bloques lógicos configurables son agrupaciones pequeñas y regulares de elementos lógicos que se establecen en una matriz bidimensional en la estructura PL y se conectan a otros recursos similares a través de interconexiones programables. Cada CLB se coloca al lado de una matriz de conmutación y contiene dos segmentos lógicos (slices), como se muestra en la Figura 2.14.

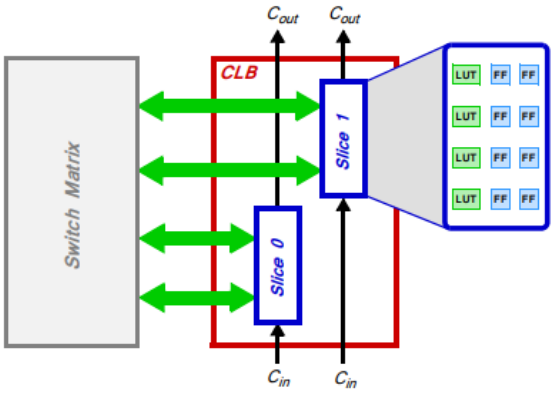


Figura 2.14. *Composición de un Bloque Lógico Configurable (CLB) [7].*

- ***Slice:*** subunidad dentro del CLB que contiene recursos para implementar circuitos lógicos combinacionales y secuenciales. Como se indica en la Figura 2.14, los slices de Zynq se componen, fundamentalmente, de 4 tablas de búsqueda (LUT) y 8 Flip-Flops (FF).

- ***Tabla de búsqueda (LUT):*** es un recurso flexible capaz de implementar:

  – Una función lógica de hasta seis entradas.
  – Una pequeña memoria de solo lectura (ROM).
  – Una pequeña memoria de acceso aleatorio (RAM).
  – Un registro de desplazamiento.

  Las LUTs se pueden combinar para formar funciones lógicas, memorias o registros de desplazamiento más grandes, según sea necesario.

- ***Flip - flop (FF):*** elemento de circuito secuencial que implementa un registro de 1 bit, con funcionalidad de restablecimiento. Uno de los FF se puede usar opcionalmente para implementar un latch.

- ***Matriz de conmutación:*** una matriz de conmutación se encuentra al lado de cada CLB y proporciona la capacidad de interconectar de forma flexible los CLBs entre sí o con otros recursos del PL.

- ***Bloque de E/S (IOB):*** los IOB son recursos que proporcionan una interfaz entre los recursos lógicos de PL y los 'pads' de dispositivos físicos utilizados para conectarse a circuitos externos. Cada IOB puede manejar una señal de entrada o salida de 1 bit y se ubican generalmente alrededor del perímetro del dispositivo.

Aunque es útil para el diseñador tener un conocimiento de la estructura subyacente del PL, en la mayoría de los casos no es necesario manejar específicamente a estos recursos ya que las herramientas Xilinx deducirán automáticamente las LUT, FF, IOB, etc. requeridas del diseño, y los maperá en consecuencia.

Recursos especiales: DSP48E1s y BRAMs

Además de la estructura general, hay dos componentes de propósito especial: Bloques de memoria (BRAM) y slices DSP48E1 para aritmética de alta velocidad. Ambos recursos están integrados en la matriz lógica en una disposición de columnas, incrustados en la lógica de la estructura y normalmente cerca uno del otro, ya que el cómputo intensivo y el almacenamiento de datos en la memoria a menudo son operaciones estrechamente asociadas.

Estos recursos son fundamentales para la aceleración hardware de algoritmos de procesado de imagen y video que nos proponemos. Esto es, como ya hemos dicho, al diseñar con Zynq, tiene sentido identificar funciones computacionalmente costosas y susceptibles de paralelizarse e implementarlas en la sección PL del dispositivo. De esta manera, la parte PL se puede utilizar para acelerar los algoritmos que residen en el PS. Se pueden encontrar más detalles sobre el DSP48E1 en [12], y sobre el BRAM en [13].

Interfaces PS - PL

Las ventajas de los dispositivos Zynq no radican solo en las propiedades de sus partes constituyentes, PS y PL, sino en la capacidad de usarlos en conjunto para formar sistemas completos e integrados. La clave de esto son las interconexiones e interfaces AXI que forman el puente entre las dos partes, además de otros tipos de conexiones entre PS y PL.

Estándar AXI

AXI deriva de las siglas Interfaz Avanzada Extensible (del inglés *Advanced eXtensible Interface).* Es un protocolo que forma parte del estándar abierto ARM AMBA [8] y la versión utilizada en el sistema es AXI4, aunque su actual versión es AXI5. El estándar AMBA fue desarrollado por ARM para uso en microcontroladores en 1966 y, desde entonces, el estándar ha sido revisado y extendido. Hoy en día se centra en sistemas en un chip (SoCs), incluyendo SoCs basados en FPGAs o, en el caso de Zynq, dispositivos que integran un procesador y una estructura FPGA

Muchos dispositivos y bloques IP de distinta procedencia están basados en este estándar. Xilinx contribuyó significativamente en la definición de AXI como una tecnología de interconexión óptima para usarse dentro de sus arquitecturas FPGA [14], [15].
Los buses AXI se usan, generalmente, para conectar el procesador (o procesadores) y otros bloques hardware implementados en el PL.
Existen tres tipos de AXI4, cada uno de los cuales representa un protocolo de bus diferente:

- ***AXI4*** [8]: se usa en interfaces asignadas a memoria (memory-mapped), permitiendo operaciones en modo ráfaga de hasta 256 ciclos de transferencia de datos con una sola fase de dirección. Proporciona el rendimiento más alto.

- ***AXI4-Lite*** [8]: soporta una única transferencia de datos por operación de lectura/escritura, sin ráfagas. Es también memory-mapped, transfiriendo en este caso un solo dato desde/hacia una dirección de memoria.

- ***AXI4-Stream*** [16]: para transmisión de datos punto a punto a alta velocidad. Admite transferencias en ráfaga de tamaño ilimitado, sin mecanismo de direccionamiento. Este tipo de bus es el más adecuado para dirigir el flujo de datos entre un origen y un destino (sin asignación de memoria).

Interconexiones e interfaces AXI

La comunicación principal entre la lógica programable (PL) y el sistema de procesamiento (PS) es a través de un conjunto de nueve interfaces AXI, cada una de las

cuales está compuesta por múltiples canales. Estos establecen conexiones dedicadas entre PL y los bloques de interconexión dentro del PS, como se indica en la Figura 2.15. Es útil definir brevemente dos términos importantes:

- ***Interconexión*:** una interconexión es efectivamente un conmutador que administra y dirige el tráfico entre las interfaces AXI conectadas. Hay varias interconexiones dentro del PS, algunas que están directamente conectadas a la PL y otras que son solo para uso interno. Las conexiones entre estas interconexiones también se forman utilizando interfaces AXI.

- ***Interfaz*:** conexión punto a punto para pasar datos, direcciones y señales de intercambio entre maestros y esclavos del sistema.

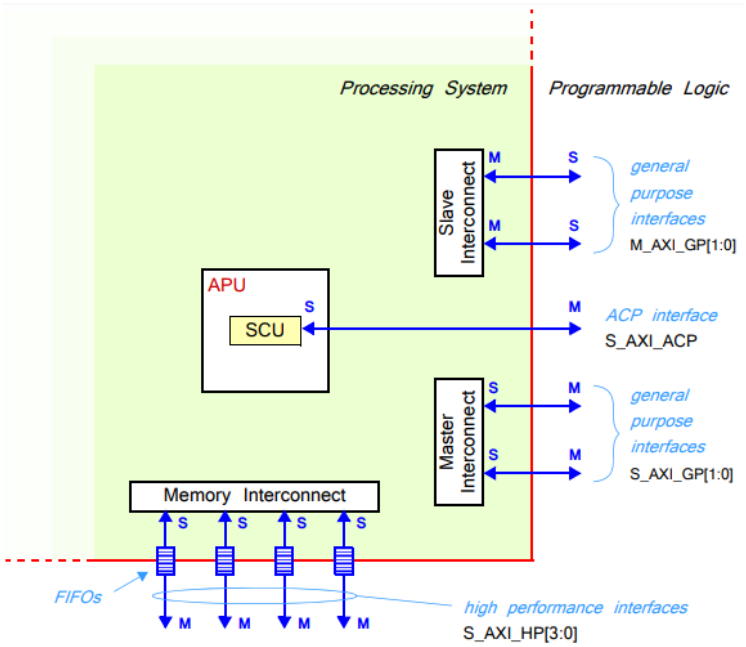


Figura 2.15. *Estructura de interconexiones e interfaces AXI conectando PS y PL [11].*

La Tabla 3 muestra un resumen de las interfaces mostradas en la Figura 2.15. Se proporciona una breve descripción de cada interfaz, y se indica el maestro y el esclavo. La convención de nomenclatura de la interfaz es indicar el papel de PS, es decir, "M" es la primera letra donde el PS es el maestro, y "S" es la primera letra donde PS es el esclavo.

Tabla 3. *Interfaces entre PS y PL [11].*

Interface Name	Interface Description	Master	Slave
M_AXI_GP0	General Purpose (AXI_GP)	PS	PL
M_AXI_GP1		PS	PL
S_AXI_GP0	General Purpose (AXI_GP)	PL	PS
S_AXI_GP1		PL	PS
S_AXI_ACP	Accelerator Coherency Port (ACP), cache coherent transaction	PL	PS
S_AXI_HP0	High Performance Ports (AXI_HP) with read/write FIFOs. (Note that AXI_HP interfaces are sometimes referred to as AXI Fifo Interfaces, or AFIs).	PL	PS
S_AXI_HP1		PL	PS
S_AXI_HP2		PL	PS
S_AXI_HP3		PL	PS

A continuación se describen con más detalle los diferentes tipos de interfaces:

- ***AXI de Propósito general (AXI_GP):*** bus de datos de 32 bits, adecuado para comunicaciones de baja y media velocidad entre el PL y el PS. La interfaz es directa y no incluye buffering. Hay cuatro interfaces de propósito general en total: el PS es el maestro de dos y el PL es el maestro de las otras dos.
- ***Puerto de Coherencia del Acelerador (ACP):*** conexión asíncrona entre el PL y la SCU dentro de la APU, con un ancho de bus de 64 bits. Este puerto se utiliza para lograr coherencia entre las memorias caché de la APU y los elementos dentro del PL. El PL es el maestro.
- ***Puertos de alto rendimiento (AXI_HP):*** las cuatro interfaces AXI de alto rendimiento incluyen buffers FIFO para acomodar el comportamiento de lectura y escritura "en ráfaga" y admiten comunicaciones de alta velocidad entre PL y los elementos de memoria en el PS. El ancho de los datos es de 32 o 64 bits, y el PL es el maestro de las cuatro interfaces.

Otras señales que cruzan el límite de PS-PL incluyen temporizadores *watchdog*, señales de reinicio, interrupciones y señales de interfaz DMA.

2.2.2. HDMI

El puerto HDMI está conectado a determinados pines de la lógica programable y tiene capacidad de entrada y salida. En nuestro sistema se utiliza como entrada para introducir un flujo de video (streaming) compatible con HDMI o DVI. La decodificación del flujo de video HDMI / DVI debe implementarse en lógica.

HDMI y DVI son protocolos serie síncronos de alta velocidad. Se requiere que las implementaciones en FPGA usen ciertas primitivas incorporadas para sintetizar la frecuencia de reloj correcta, serializar la transmisión y mantener un bloqueo en la señal.

El protocolo HDMI / DVI utiliza TMDS (Señalización Diferencial con Transición Minimizada) como estándar de E / S y es compatible con Zynq mediante los buffers de E / S de la lógica programable.

Es importante tener precaución a la hora de conectar dispositivos HDMI/DVI. No se deben conectar dispositivos HDMI / DVI con alimentación a la placa ZYBO sin alimentar.

2.2.3. VGA

Como salida de imagen y vídeo de nuestro sistema se utiliza el puerto VGA. La placa ZYBO usa 18 pines de la lógica programable para crear un puerto de salida VGA analógico, traduciéndose en una profundidad de color de 16 bits y dos señales de sincronización estándar (HS - Sincronización Horizontal, y VS - Sincronización Vertical). Con 5 bits cada uno para rojo y azul y 6 bits para verde, se pueden mostrar 65,536 (32 × 32 × 64) colores diferentes, uno para cada patrón único de 16 bits.

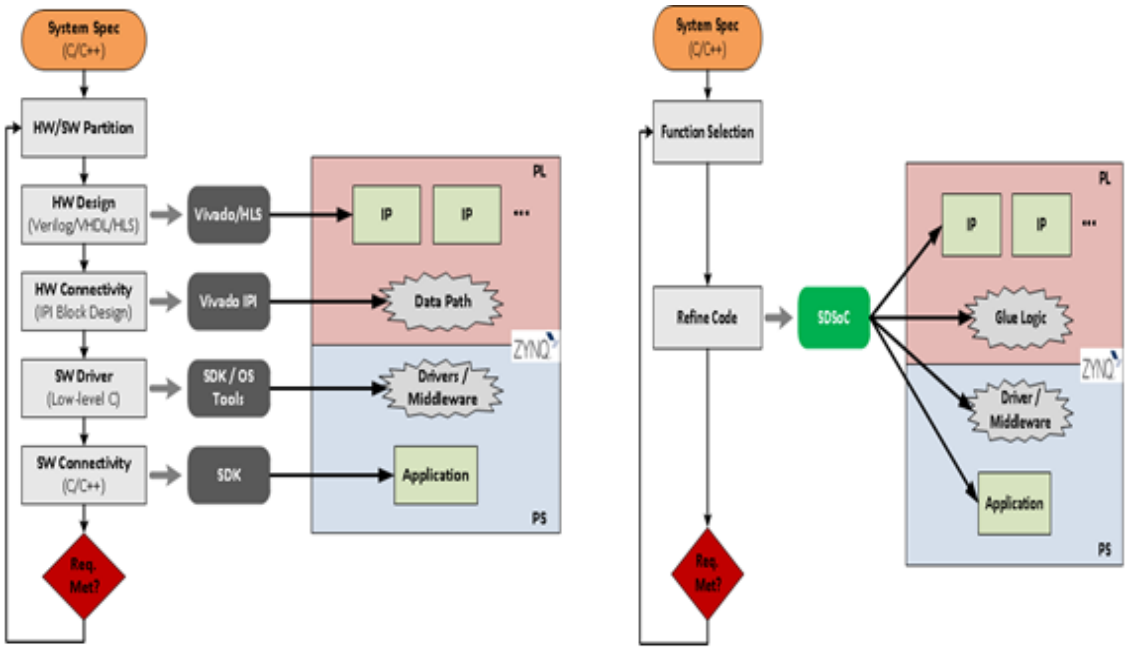
Se debe implementar un circuito en la lógica programable para controlar las señales de sincronización y color con la temporización correcta para producir un sistema de salida funcional.

Las señales de temporización VGA están especificadas, publicadas, con derecho de autor y comercializadas por la organización VESA. En el manual de referencia ZYBO [6] se puede consultar un ejemplo de temporización de un sistema con monitor VGA en modo 640 x 680.

2.3. Herramientas de diseño

En este apartado se describe el flujo de diseño de sistemas hardware-software convencional y las herramientas de Xilinx que lo soportan, y se introduce el entorno de co-diseño que se ha explorado en este trabajo, poniendo de manifiesto, fundamentalmente, las diferencias conceptuales entre ambas aproximaciones.

La Figura 2.16 (a) resume el flujo de codiseño convencional. En esta metodología el diseñador comienza por determinar qué partes del sistema se implementan en hardware y qué partes se implementan en software (HW/SW partition). A partir de aquí, el diseño del hardware y del software se abordan secuencialmente. En primer lugar es necesario desarrollar los bloques hardware necesarios que se hayan identificado en la fase de particionamiento y que no estén disponibles en las librerías de Xilinx (Hardware Design). Convencionalmente se utiliza para esto una metodología RTL a hardware. Esto es, la funcionalidad se describe a nivel RTL usando Verilog o VHDL y se sintetiza e implementa en el entorno Vivado. Vivado también permite añadir las interfaces necesarias para poder integrar el bloque hardware diseñado como periférico de un sistema con procesador.



a) Flujo de codiseño convencional. **b)** Flujo de codiseño SDSoC.

Figura 2.16. *Flujos de codiseño hardware/software.*

Más recientemente, Xilinx ha introducido una herramienta con la que el diseño de los bloques hardware comienza con una descripción a nivel de algoritmo en un lenguaje de alto nivel, a partir de la cual se genera la descripción RTL que luego se sintetiza en Vivado. Esto es, se automatiza también el diseño de la solución a nivel RTL, lo que se conoce como síntesis de alto nivel o HLS, del inglés High Level Synthesis.

El siguiente paso es diseñar la plataforma hardware con el procesador y los periféricos necesarios de acuerdo con el particionamiento hardware-software (HW Connectivity). Esta tarea se realiza usando Vivado IPI (IP Integrator). En Vivado IPI se configuran y conectan los distintos bloques hardware que conforman la plataforma, incluido el procesador.

El desarrollo del software se realiza en el entorno SDK, del inglés Software Development Kit. Requiere tanto el diseño de los drivers software para los periféricos, que implica una programación en lenguaje C de bajo nivel (SW Driver), como el desarrollo de la aplicación que se ejecuta en el procesador (SW Connectivity).

La Figura 2.16 (b) muestra el flujo de codiseño con SDSoC, del inglés Software Defined System on Chip. El sistema completo se desarrolla en un único entorno de diseño, integrando completamente el diseño del hardware y del software. La herramienta además asiste en la fase de particionamiento y libera al diseñador de determinadas tareas de programación de bajo nivel.

Hemos considerado conveniente describir brevemente las herramientas y entornos de diseño mencionados. Las más novedosas y que son objeto de exploración de este trabajo se describirán posteriormente con más detalle.

2.3.1. Vivado IDE

El entorno de diseño integrado (IDE) de Vivado® proporciona una interfaz gráfica de usuario (GUI) intuitiva, que facilita el recorrido del flujo de diseño RTL a bitstream utilizando las funcionalidades integradas, que incluyen:

- Integración de IPs.
- Simulación de comportamiento, funcional y temporal.
- Síntesis RTL.
- Síntesis física (place and route).
- Análisis de potencia.
- Definición de restricciones físicas y temporales
- Análisis temporal estático.
- Generación del fichero de programación de la FPGA (bitstream).

Vivado mantiene una base de datos unificada durante todo el proceso de diseño que permite cargar la lista de componentes y conexiones (netlist), asignar restricciones al diseño y aplicarlas al dispositivo de destino. Esto proporciona la capacidad de visualizar e interactuar con el diseño en cada etapa, haciendo posible por ejemplo, aplicar diferentes opciones de síntesis e implementación para conseguir determinados objetivos de diseño (temporales, de recursos o de potencia) o realizar simulaciones para validar el diseño.

Entre los componentes del entorno integrado de Vivado se encuentra el navegador de flujo (Flow Navigator), que proporciona acceso a los comandos y herramientas para desarrollar el diseño desde el inicio hasta la creación del bitstream. A medida que se ejecutan estos comandos y herramientas, se actualizan los datos de diseño, las ventanas gráficas y las ventanas de resultados. Las diferentes secciones en el Flow Navigator permiten realizar las siguientes acciones:

- **Project Manager:** cambiar la configuración general, agregar o crear fuentes [17], ver plantillas de lenguajes de descripción hardware y abrir el catálogo de Vivado IP [18].
- **IP Integrator:** crear, abrir o generar un diseño de bloques [19].
- **Simulation:** cambiar la configuración de la simulación o simular el diseño activo [20].

- **RTL Analysis:** abrir un diseño elaborado, ejecutar verificaciones de reglas de diseño (DRCs) y generar un esquema RTL [17].
- **Synthesis:** cambiar la configuración de síntesis, sintetizar el diseño activo o abrir el diseño sintetizado [21].
- **Implementation:** cambiar la configuración de la implementación, implementar el diseño activo o abrir el diseño implementado [22].
- **Program and Debug:** cambiar la configuración del bitstream, generar un archivo bitstream, abrir una sesión de hardware en el IDE de Vivado e iniciar el analizador lógico de Vivado [23].

Un aspecto importante a tener en cuenta es que Vivado IDE solo admite diseños que utilicen FPGAs de la serie 7 de Xilinx. Si se dese ampliar información sobre el resto de componentes del IDE de Vivado se puede consultar [24].

2.3.2. Vivado HLS

La herramienta Vivado High-Level Synthesis (HLS) transforma la especificación de un algoritmo C/ C++ o System C, en una implementación RTL sintetizable. SDSoC, que será introducido a continuación, la usa de manera interna. Es conveniente conocer los beneficios que aporta la síntesis de alto nivel y las fases que comprende con el fin de sentar las bases sobre el funcionamiento interno de la herramienta SDSoC.

La síntesis de alto nivel une los dominios de hardware y software, proporcionando los siguientes beneficios principales:

- Mejora la productividad. Desarrollar algoritmos a nivel C reduce significativamente los tiempos de diseño frente al desarrollo a nivel de RTL.
- Permite la exploración rápida del espacio de soluciones a nivel arquitectural mediante la aplicación de directivas de optimización que guían el proceso de síntesis, lo cual aumenta la probabilidad de encontrar una implementación óptima.
- Posibilita el diseño de módulos IP.

La Figura 2.17 ilustra el flujo de diseño correspondiente a la síntesis de alto nivel empleando Vivado HLS.

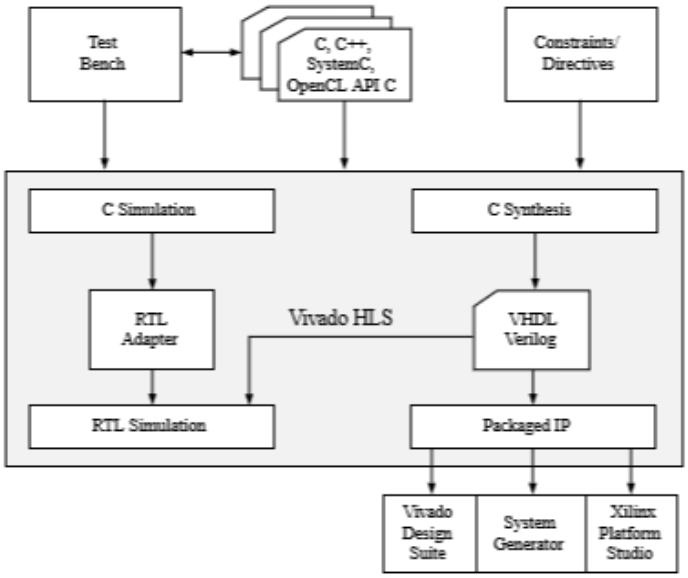


Figura 2.17. *Flujo de diseño de la herramienta Vivado HLS [21].*

Los pasos en el flujo de diseño de Vivado HLS se pueden resumir como sigue:

1. Escribir la descripción del bloque hardware a nivel algoritmo en C, C++ o System C
2. Validar la descripción. Es lo que se denomina C-Simulation. Se trata de validar el código que hemos escrito. Requiere escribir un testbench. Básicamente el código se compila y se ejecuta.
3. Sintetizar la descripción RTL. Esta fase se describe con más detalle a continuación.
4. Verificar la implementación RTL. Se simula la solución RTL generada y se compara los resultados con los obtenidos por la descripción de alto nivel. No es necesario desarrollar un testbench para está simulación RTL. La propia herramienta lo genera a partir del utilizado para la validación del código inicial. También la comparación es transparente para el usuario.
5. Empaquetar la implementación RTL como IP para su uso en distintas herramientas.

La síntesis de alto nivel incluye las siguientes fases:

- **Planificación (Scheduling):** Determina qué operaciones del algoritmo ocurren durante cada ciclo de reloj. La planificación se realiza en base a la frecuencia de operación objetivo, el dispositivo sobre el que se va a implementar el diseño, las directivas/restricciones del usuario y el propio código.

  Así, si el período de reloj es más largo o si se utiliza una FPGA más rápida, se completan las operaciones requeridas para implementar una determinada tarea en un ciclo de reloj o, incluso, se pueden encadenar operaciones en un mismo

ciclo. Por el contrario, si el período de reloj es más corto o se emplea una FPGA más lenta, la síntesis de alto nivel automáticamente distribuye las operaciones necesarias a lo largo de más ciclos de reloj, por lo que algunas operaciones pueden ser Implementadas como recursos multiciclo.

- **Unión (Binding):** Determina qué recurso hardware de la librería (cores) implementa cada operación. Para implementar la solución óptima, la síntesis de alto nivel utiliza información sobre el dispositivo de destino y tiene en cuenta las directivas de usuario.
- **Extracción de la lógica de control:** Extrae la lógica de control para crear una máquina de estados finitos (FSM) que secuencia las operaciones en el diseño RTL. La funcionalidad del control viene determinada, básicamente, por los bucles y construcciones condicionales del código.

La herramienta HLS sintetiza los distintos elementos del código como sigue:

- Las funciones se sintetizan como bloques hardware en la jerarquía RTL. Si el código C incluye una jerarquía de subfunciones, el diseño RTL final incluye una jerarquía de módulos o entidades que tienen una correspondencia con la jerarquía de la función original. Por defecto, los distintos bloques hardware no operan en paralelo. Usando directivas de optimización puede controlarse la jerarquía de diseño resultante y forzar la operación concurrente de los distintos bloques.
- Los argumentos de las funciones se sintetizan en puertos de entrada/salida del correspondiente bloque hardware. Para descripciones C y C++ está síntesis de la interfaz se realiza automáticamente. Es decir se sintetizan no sólo las operaciones del algoritmo sino el hardware necesario para la lectura y escritura de los puertos.
- Los bucles en las funciones se mantienen "enrollados" por defecto. La síntesis crea la lógica para una iteración del bucle, y el diseño RTL ejecuta esta lógica para cada iteración del bucle secuencialmente. Una iteración no comienza hasta que haya terminado la anterior. Usando directivas de optimización se pueden "desenrollar" parcial o totalmente los bucles o forzar a que una iteración no tenga que esperar a que acabe la anterior para comenzar. En ambos casos se está paralelizando las operaciones asociadas al bucle correspondiente. Esto se traduce en una disminución de la latencia e incremento del rendimiento del módulo sintetizado, lo que permite que varias o todas las iteraciones ocurran en paralelo.
- Los arrays se sintetizan como memorias. Por defecto se mapean a memorias RAM, aunque utilizando directivas se puede forzar el uso de registros o FIFOs y así adecuar la arquitectura de memoria al algoritmo con el objetivo de evitar el cuello de botella que muchas veces supone el acceso a memoria.

En general la fase de síntesis es un proceso iterativo en el que para un mismo código se van generando distintas soluciones RTL, aplicando directivas de síntesis, con el objetivo de cumplir nuestras restricciones de diseño u optimizar la implementación de acuerdo con determinados criterios. Así, el análisis de las soluciones obtenidas para identificar qué está limitando las prestaciones del diseño o qué recursos utiliza es crucial. Con este fin, HLS proporciona informes y herramientas de análisis en base a los cuales el diseñador determina qué directivas o acciones debe aplicar en cada momento. El informe de síntesis contiene la siguiente información:

- *Área*: Recursos hardware del FPGA necesarios para implementar el bloque hardware sintetizado, incluyendo LUTs, registros, BRAM y sDSP48s.
- *Latencia*: Número de ciclos de reloj requeridos por el bloque hardware para procesar los datos de entrada.
- *Intervalo de Iniciación (II)*: Números de ciclos de reloj antes de que el bloque hardware pueda aceptar nuevos datos de entrada.
- *Latencia de iteración de bucle*: Número de ciclos de reloj que se requieren para completar una iteración del bucle.
- *Intervalo de inicio del bucle*: Número de ciclos de reloj antes de que comience la siguiente iteración del bucle para procesar los datos.
- *Latencia de bucle*: Número de ciclos para ejecutar todas las iteraciones del bucle.

En base al análisis realizado el usuario puede, a modo de ejemplo, aplicar directivas para:

- Incrementar el paralelismo o la concurrencia de operaciones para reducir la latencia y/o el intervalo.
- Rediseñar las memorias para evitar cuellos de botella asociados a accesos a los datos almacenados.
- Limitar los recursos hardware asociados a determinadas operaciones.
- Forzar la compartición de recursos hardware entre distintas operaciones.
- Modificar el código para refinar el número de bits.

2.3.3. SDSoC

El entorno SDSoC (Software-Defined System-on-Chip) es un conjunto de herramientas, englobadas bajo un entorno de desarrollo integrado (IDE) basado en Eclipse, que permite desarrollar sistemas integrados heterogéneos utilizando los dispositivos Zynq-7000 y Zynq UltraScale. La entrada principal es una aplicación C / C ++, en la que se pueden seleccionar una o más funciones (que denominaremos funciones hardware)

para ser implementadas en la lógica programable con propósito de aceleración. La generación del sistema hw/sw se puede dirigir insertando directivas de usuario en el código fuente.

SDSoC aumenta el nivel de abstracción del co-diseño hw/sw, permitiendo un flujo de trabajo a alto nivel. Este nivel de abstracción se consigue gracias al compilador que incorpora el entorno, que genera de forma automática un sistema híbrido a partir de una especificación C/C++.

El compilador de SDSoC se dirige a una plataforma hardware determinada e invoca la herramienta Vivado HLS para “compilar” las funciones C / C ++ seleccionadas para ser implementadas en la PL. Por tanto, genera un sistema hardware/software completo. Además se genera el fichero de programación de la FPGA (bitstream) invocando las herramientas de Vivado Design Suite.

Cuando se inicia un nuevo diseño en SDSoC, el diseñador debe seleccionar una plataforma hardware específica. Además, el entorno soporta varios sistemas operativos para generar la aplicación: Standalone (o bare-metal), Linux y FreeRTOS [25].

En definitiva, SDSoC simplifica el desarrollo de aplicaciones integradas que incorporan aceleradores de hardware.

El siguiente capítulo introduce conceptos relacionados con el desarrollo de plataformas hardware para SDSoC y describe la plataforma hardware base para el procesado de imagen y video que vamos a utilizar.

3. Diseño de la Plataforma Hardware para SDSoC

Una plataforma SDSoC define una arquitectura base de hardware y software, organizada mediante un sistema de archivos raíz (root). La definición hardware incluye los recursos disponibles para la herramienta, como los relojes y su frecuencia, interfaces de memoria externa, puertos e interrupciones AXI, entrada / salida personalizada y controladores para los periféricos de la plataforma. Por otro lado, la definición software incluye la localización de los componentes del sistema operativo, librerías software y directorios dentro del sistema de archivos raíz de la plataforma.

El entorno SDSoC incluye diferentes plataformas base para el desarrollo de aplicaciones sobre dispositivos Zynq-7000 AP SoC (ZC702, ZC706, MicroZed, ZedBoard y Zybo) y Zynq UltraScale+ MPSoC (ZCU102). Además, permite el uso de plataformas personalizadas o proporcionadas por otros fabricantes, diseñadas utilizando la herramienta IP Integrator disponible en el IDE de Vivado.

3.1. Arquitecturas para el procesado de imagen y vídeo

Los sistemas de procesado de imagen y video implementados como SoCs sobre dispositivos Zynq comúnmente siguen una de las dos arquitecturas genéricas siguientes: transmisión directa del flujo de vídeo (video stream) y transmisión mediante frame-buffer [26].

La forma más sencilla y eficiente de procesar vídeo suele ser una arquitectura de transmisión directa como la que se ilustra en la Figura 3.1.

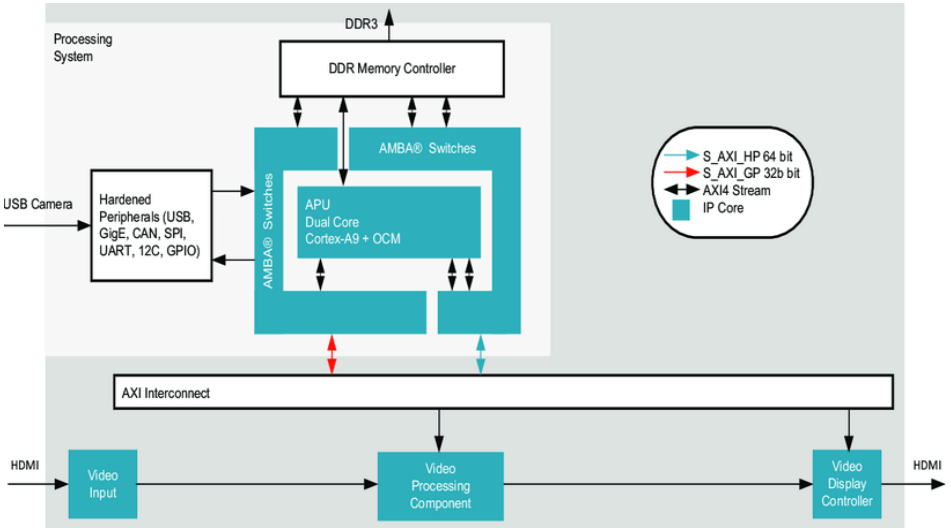


Figura 3.1. *Arquitectura de transmisión directa de flujo de vídeo en Zynq [26].*

En esta arquitectura, los datos de píxeles llegan a los pines de entrada de la lógica programable y se transmiten directamente a un componente de procesado de video, para luego ser transmitidos directamente a la salida de video. Esto requiere que todas las etapas de procesado estén sincronizadas para que no se produzca la pérdida de ningún dato.

Por otro lado, en una arquitectura frame-buffer los píxeles entrantes se almacenan en un buffer localizado en la memoria DDR disponible en la placa de desarrollo y a continuación se extraen para ser procesados. Seguidamente, son almacenados de nuevo en la memoria externa para ser emitidos en la salida a través de un controlador de visualización de video. La Figura 3.2 muestra un diagrama de bloques de esta arquitectura implementada sobre un dispositivo Zynq.

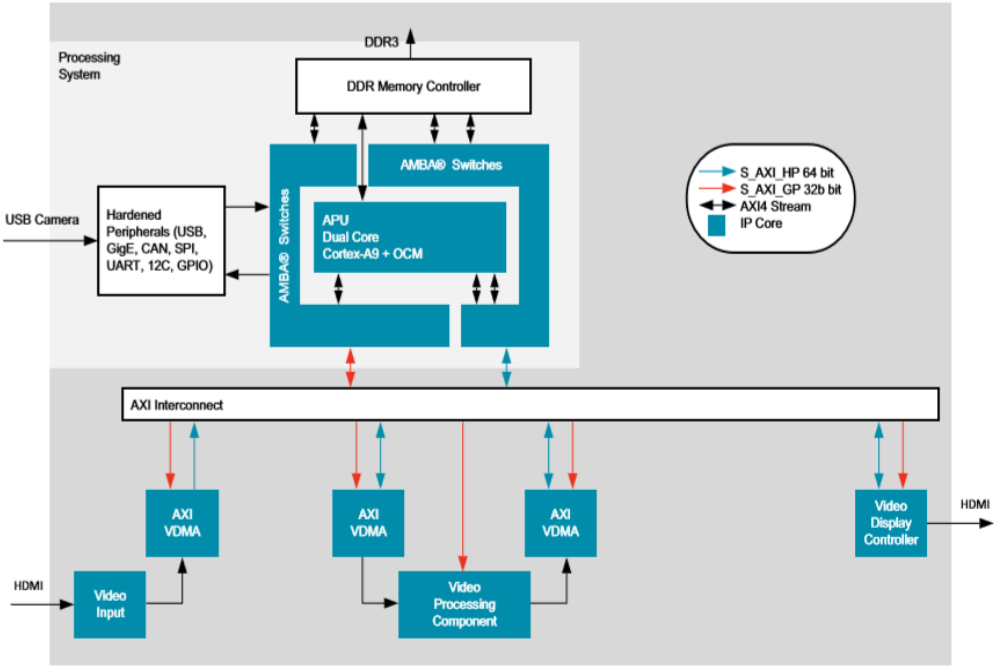


Figura 3.2. ***A****rquitectura de transmisión frame-buffer en Zynq [26].*

Esta arquitectura permite una mayor disociación entre la velocidad de video y la velocidad de procesamiento, aunque requiere suficiente ancho de banda de memoria para leer y escribir los frames de video en la memoria externa. Esta será la arquitectura en la que se basa el sistema de procesado de vídeo.

3.2. Desarrollo de la plataforma hardware

En general, cualquier diseño dirigido a un dispositivo Zynq-7000 creado mediante Vivado IP integrator puede ser la base de una plataforma SDSoC. El proceso de desarrollo de la plataforma hardware SDSoC es conceptualmente sencillo:

1. Diseñar el sistema hardware usando Vivado Design Suite.
2. Cargar las interfaces de programación de aplicaciones (API-Application Programming Interface) de SDSoC.
3. Ejecutar los scripts Tcl que definen dichas APIs en la consola Tcl de Vivado para llevar a cabo los siguientes pasos:
    a. Declarar el nombre de la plataforma hardware
    b. Incluir una breve descripción de la plataforma.
    c. Declarar los puertos de reloj de la plataforma.
    d. Declarar las interfaces de bus AXI en la plataforma.
    e. Declarar las interfaces de bus AXI4-Stream de la plataforma.
    f. Declarar las interrupciones disponibles en la plataforma.
    g. Generar el archivo de metadatos de la descripción del hardware de la plataforma.

Como punto de partida para el desarrollo de este trabajo, se tomó como referencia un diseño proporcionado por la empresa Digilent, conteniendo los drivers necesarios para el control de los distintos relojes que intervienen en el sistema y la monitorización vía VGA. Esta plataforma fue diseñada para su funcionamiento en la versión SDSoC 2015.4. Por este motivo, entre los objetivos iniciales del trabajo se planteó la necesidad de generar una plataforma personalizada para la placa ZYBO compatible con versiones más actualizadas de SDSoC.

El desarrollo de la nueva plataforma base se lleva a cabo mediante IP integrator, dentro del entorno Vivado Design Suite, siguiendo los pasos citados anteriormente. Esta plataforma está dirigida al dispositivo ZYBO y permitirá al entorno SDSoC conectar el componente de procesado de vídeo con el resto de elementos de la plataforma base.

El diseño consta de los siguientes módulos IP:

- DVI to RGB Video Decoder [27].
- Video In to AXI4-Stream [5].
- Video Timing Controller [4].
- Dynamic Clock Generator.
- AXI Video Direct Memory Access [28].
- AXI Interconnect [29].
- Zynq Processing System [30].
- Axi4-Stream to Video Out [5].
- RGB to VGA output [31].
- Processor System Reset Module [32].
- Concat [33].
- Slice [12].

A continuación, se describe el flujo de datos de la plataforma.

La placa ZYBO recibe un flujo de video de entrada a través del puerto HDMI. Este flujo se decodifica mediante el módulo IP "DVI to RGB", el cual genera los datos de video en formato RGB de 24 bits, junto con el reloj de píxeles y las señales de sincronización recuperadas del enlace TMDS. Este bloque está controlado por un reloj del PS a 200Mhz

Los datos se transmiten a continuación hacia el frame-buffer mediante el protocolo AXI4-Stream. Para ello, los datos de vídeo en formato RGB pasan a través del bloque "IP Video In to AXI4-Stream" proporcionando, en conjunto con el Controlador de Sincrinización de Vídeo (VTC), una interfaz entre la fuente de vídeo y el protocolo de video AXI4-Stream. La entrada del controlador VTC detecta automáticamente los pulsos de sincronización horizontal y vertical, la polaridad, el tiempo de borrado y los píxeles de video activos.

La elección de un frame-buffer para el sistema, basado en AXI4-Stream, es el bloque AXI-VDMA. Este bloque es compatible con las interfaces de video AXI4-Stream [5] de forma nativa, lo que significa que las señales de comienzo del frame (SOF) y fin del frame (EOL) son interpretadas y generadas adecuadamente. Se utiliza en el modo de registro directo simple y se accede a los registros de inicialización, estado y administración en el módulo AXI VDMA a través de una interfaz AXI4-Lite. Además de las dos interfaces de transmisión AXI4-Stream de entrada y de salida, el bloque cuenta con dos canales de acceso a la memoria: MM2S (mapeado en memoria a stream) y S2MM (stream a mapeado en memoria).

El bloque AXI VDMA recibe los datos de entrada en forma de un flujo AXI4-Stream, escribiéndolos en la memoria DDR a través del canal S2MM. Para generar su salida, lee los datos de la memoria DDR a través del canal MM2S.

Los datos generados por el módulo AXI VDMA se transfieren al bloque Axi4-Stream to Video Out, que convierte las señales de la interfaz AXI4-Stream en una interfaz de salida de video paralelo estándar con señales de temporización. Este bloque funciona junto con el Controlador de Temporización de Video (VTC), que genera los períodos de refresco (horizontal y vertical) y los pulsos de sincronización. El ancho y el intervalo de estos pulsos se configuran a través de la interfaz AXI Lite.

El funcionamiento conjunto de los dos bloques mencionados en el párrafo anterior proporciona un puente entre los bloques de procesamiento de video con interfaces AXI4-Stream y la salida de video del sistema. Esta salida se genera a través del núcleo RGB to VGA ouput, que acepta como entrada el vídeo paralelo estándar y un bus de píxeles RGB debidamente sincronizado para ser transmitidos a través del puerto VGA.

La salida del sistema no incluye restricciones de tiempo. La frecuencia de reloj de píxel (entrada PixelClk del núcleo RGB to VGA output) viene determinada por el bloque Dynamic CLock Generator, que genera una señal de reloj configurable mediante el software de la aplicación.

La Figura 3.3 muestra el diagrama de bloques de la plataforma hardware que servirá como base para generar el sistema completo dentro del entorno SDSoC. Por simplicidad, se muestran solo las conexiones de interfaz.

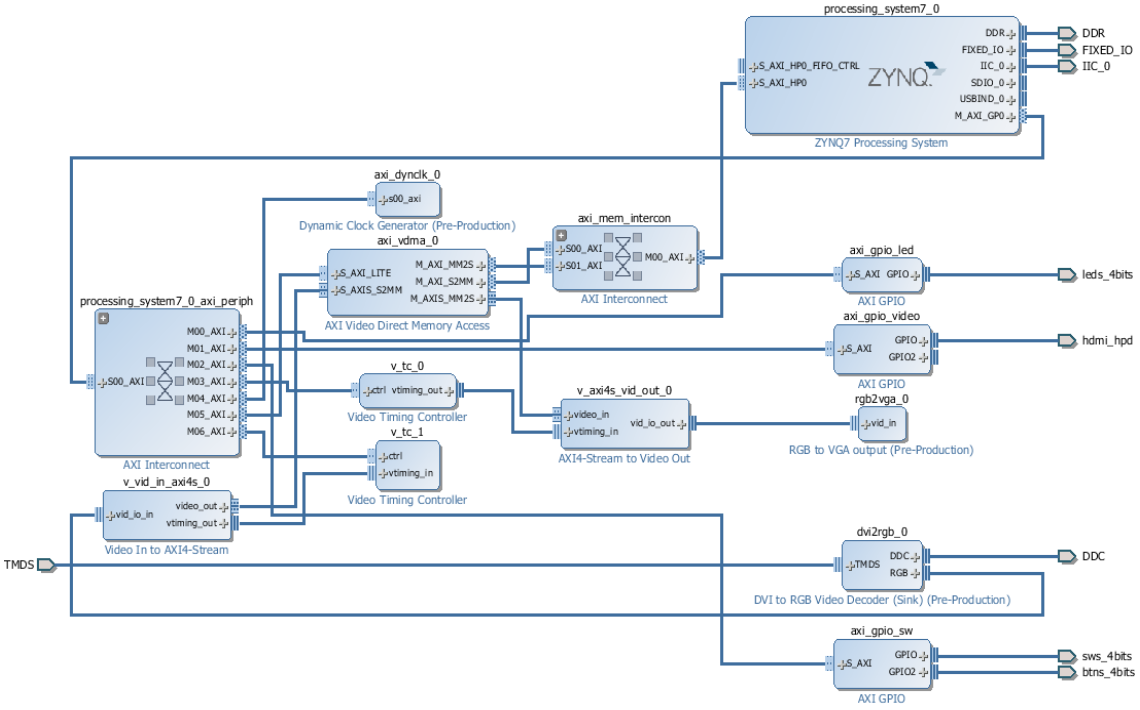


Figura 3.3. *Diagrama de bloques de la plataforma base.*

Se puede consultar el Anexo I de la memoria para analizar el diagrama de bloques completo.

3.3. Elementos y estructura de una plataforma base

Para que una plataforma base sea reconocida por SDSoC, esta debe estructurarse en un serie de directorios que contienen los elementos necesarios. En la Figura 3.4 se puede observar la estructura típica de directorios de una plataforma SDSoC.

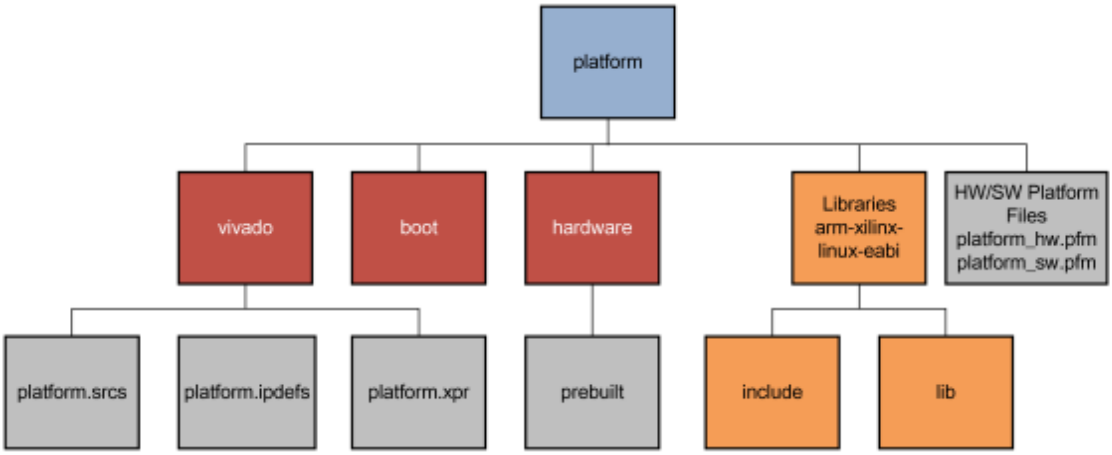


Figura 3.4. *Estructura de directorios de una plataforma SDSoC [34].*

Cada directorio consta de los siguientes elementos:

- Proyecto Vivado Design Suite [vivado]
  - Fuentes.
  - Restricciones.
  - Bloques IP.

- Archivos de arranque de Linux [boot]
  - Objetos relacionados con Linux (device-tree, u-boot, Linux-kernel, ramdisk).

- Archivos de hardware predeterminados (opcional) [hardware]
  - Bitstream
  - Archivos de software de información de puertos y registro de dispositivos generados previamente.
  - Archivos de interfaz de hardware y software generados previamente.

- Librería de archivos de cabecera [Libraries arm-xilinx-linux-eabi]
  - Archivos de cabecera de biblioteca.

- Bibliotecas estáticas (opcionales).

- Archivos de metadatos [HW/SW Platform Files]
  - Archivo de descripción hardware de la plataforma (_hw.pfm), generado mediante las herramientas Vivado.
  - Archivo de descripción del software de la plataforma (_sw.pfm), escrito a mano.

3.4. Reglas de diseño

En el diseño del hardware de la plataforma, hay varias reglas que deben contemplarse:

1. El nombre del proyecto Vivado debe coincidir con el nombre de la plataforma. Si el proyecto Vivado contiene más de un diagrama de bloques, el diagrama de bloques utilizado es el que tiene el mismo nombre que la plataforma.
2. Todo módulo IP de la plataforma que no sea parte del catálogo estándar de IPs de Vivado debe ser local al proyecto Vivado Design Suite de la plataforma. No se permiten referencias a rutas externas de repositorios de IPs.
3. La plataforma debe contener un bloque IP del sistema de procesamiento (PS) disponible en el catálogo de IPs de Vivado.
4. Los puertos de interfaz del hardware de la plataforma SDSoC deben corresponder a buses AXI o AXI4-Stream, o a señales de reloj, reinicio o interrupción. Los tipos de bus personalizados u otras interfaces de hardware deben permanecer internos a la plataforma.
5. Toda plataforma debe declarar al menos un puerto maestro AXI de propósito general en el PS y un bloque "AXI Interconnect" conectado a dicho puerto, que serán utilizados por los compiladores SDSoC para configurar los IPs de los aceleradores hardware y lo bloques de transferencia de datos.
6. Toda plataforma debe declarar al menos un puerto esclavo AXI asociado al PS que los compiladores de SDSoC utilizarán para acceder a la memoria DDR desde los IPs de los aceleradores hardware y los bloques de transferencia de datos.
7. Para compartir un puerto AXI entre el entorno SDSoC y la lógica de la plataforma (por ejemplo, S_AXI_ACP), se debe emplear un AXI maestro o esclavo que no esté en uso en un bloque IP AXI Interconnect y la plataforma debe usar los puertos con índices menos significativos.
8. Cada interfaz AXI de la plataforma se conectará a un solo reloj de movimiento de datos mediante el entorno SDSoC. No obstante, las funciones del acelerador

generadas por los compiladores SDSoC pueden ejecutarse con un reloj diferente proporcionado por la plataforma.

9. Todas las interfaces AXI4-Stream de la plataforma requieren el uso de las señales de protocolo (handshake) TLAST y TKEEP para interactuar con los IPs de movimiento de datos generados por el entorno SDSoC.
10. Cada señal de reloj empleada en la plataforma debe tener asociada un bloque IP de "reset" (Processor System Reset IP) disponible en el catálogo de IPs de Vivado.
11. Las entradas de interrupción de la plataforma se deben conectar mediante un bloque Concat (xlconcat) conectado al puerto IRQ_F2P del sistema de procesamiento.

4. Implementación del sistema

En este capítulo se describe la metodología utilizada para la implementación del sistema en el entorno SDSoC, abarcando la metodología de diseño, las arquitecturas de memoria empleadas y cómo integrarlas en el algoritmo, así como la aceleración hardware de la aplicación de procesado de imagen y vídeo. Además, se presentan técnicas de optimización del acelerador hardware mediante el uso de directivas SDS y HLS, respectivamente.

4.1. SDSoC Environment

La metodología de diseño empleada para generar un sistema híbrido hw/sw en el entorno SDSoC se denomina "Software-Centric". Este entorno facilita el desarrollo de aplicaciones empotradas con aceleración hardware, empleando lenguajes de programación estándar C/C++, como ya adelantamos en el Capítulo 2.

Las funciones seleccionadas para ser aceleradas se sintetizan con HLS para ser implementadas en la lógica programable. El compilador determina el flujo de datos entre el software y las funciones hardware, generando un sistema hw/sw sobre un determinado sistema operativo (bare-metal, Linux o FreeRTOS).

4.1.1. Metodología de diseño

El desarrollo de un sistema hw/sw con SDSoC requiere la selección de una plataforma base, que será “personalizada” con redes de movimiento de datos y aceleradores hardware específicos de la aplicación. La plataforma diseñada en el capítulo anterior es la plataforma seleccionada para generar nuestro sistema.

Con la plataforma base seleccionada, la metodología de diseño con SDSoC, representada en la Figura 4.1, se resume en los siguientes pasos:

1. Desarrollar la aplicación en lenguaje de alto nivel (C/C++).
2. Identificar la función (o funciones) que va a ser implementada en la lógica programable.

3. Analizar el bloque (o bloques) hardware sintetizado utilizando los informes de estimación de rendimiento y de estimación de recursos generados por la herramienta.
4. Optimizar la transferencia de datos y el paralelismo mediante el uso de directivas de usuario (pragmas SDS).
5. Optimizar el acelerador empleando directivas HLS.
6. Analizar de nuevo los informes de estimación y, si es necesario, realizar nuevas optimizaciones.
7. Generar el sistema y ejecutar la aplicación en la placa de desarrollo.

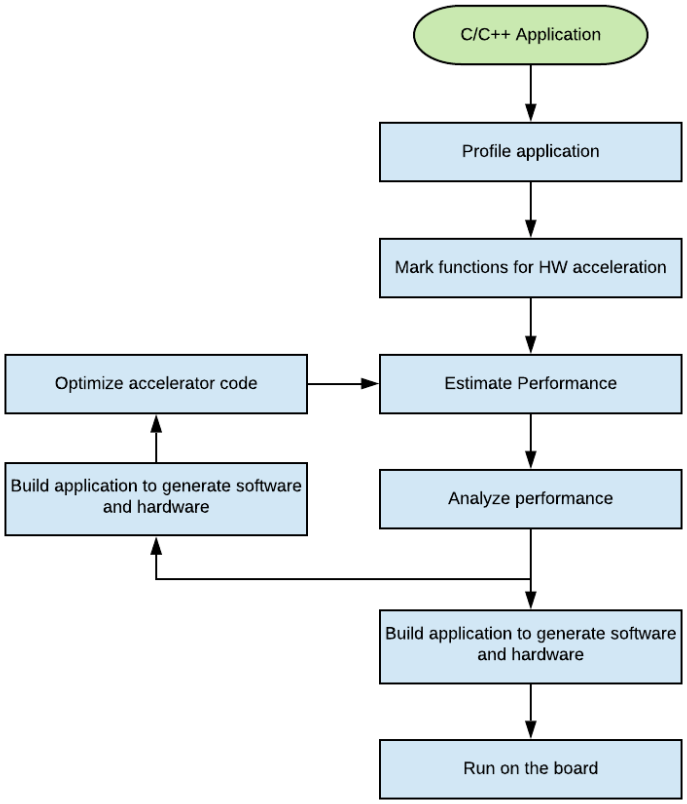


Figura 4.1. *Metodología de diseño con SDSoC.*

4.1.2. Red de movimiento de datos

SDSoC implementa mediante aceleradores hardware la función o funciones seleccionadas e incorpora una "red de movimiento de datos" para el intercambio de información entre el PS y estos aceleradores y entre aceleradores. Para guiar a la herramienta a la hora de generar dicha red, se pueden insertar pragmas en su código fuente, denominados "pragmas SDS" para distinguirlos de los empleados con la herramienta Vivado HLS (pragmas HLS). Si no se utiliza ningún pragma SDS, SDSoC genera dicha red basándose únicamente en un análisis del código fuente.

A continuación se describen los componentes de la red de movimientos de datos que genera SDSoC y cómo se puede modificar mediante el uso de directivas de usuario.

La red de movimiento de datos en consta de tres componentes: puertos del sistema de memoria ubicados en el sistema de procesado (A), data movers entre el PS y los aceleradores (B) y la interfaz del acelerador hardware (C). La Figura 4.2 ilustra estos componentes.

Figura 4.2. *Componentes de la red de movimiento de datos.*

Puertos del Sistema (A)

La conexión entre los data mover y el PS se realiza a través de los puertos del sistema. Como se describió en el Capítulo2, el sistema de procesamiento (PS) en los dispositivos Zynq tiene tres tipos de puertos de sistema que se utilizan para transferir datos desde la memoria del procesador a la lógica programable (PL) y viceversa. Se trata de un puerto de coherencia del acelerador (ACP) que permite que el hardware acceda directamente a la memoria caché L2 del procesador de manera coherente; cuatro puertos de alto rendimiento (HP0-3), que utilizan una interfaz FIFO asíncrona (AFI) para proporcionar acceso directo a la memoria DDR o a la memoria implementada en el chip (OCM) sin pasar por la memoria caché del procesador; y dos puertos de E / S de propósito general (GP0 / GP1) que permiten al procesador leer y escribir los registros o espacios de memoria incluidos en los aceleradores hardware.

La elección del puerto del sistema que se use para un determinado acelerador depende de las características de los datos que se transfieren (atributo de memoria caché y tamaño). El compilador SDSoC las analiza y elige automáticamente el puerto de sistema que mejor se adapta para usar en cada transferencia de datos, es decir, conecta cada data mover con el puerto de sistema apropiado.

Por otro lado, SDSoC permite especificar directamente a qué puerto del sistema se conectará el data mover correspondiente a un argumento dado de la función hardware, usando el siguiente pragma:

```
#pragma SDS data sys_port(arg:port)
```

Donde port puede ser ACP o AFI.

A continuación, se muestran unas pautas generales a tener en cuenta a la hora de especificar los puertos de sistema mediante directivas de usuario:

- Si los datos se asignan en memoria empleando sds_alloc_non_cacheable () o sds_register_dmabuf (), es mejor utilizar el puerto AFI para evitar el vaciado/invalidación de caché. Si los datos se asignan de otra manera, es mejor emplear el puerto ACP para un vaciado (invalidación) rápido de la memoria caché.

- Cuando el tamaño de los datos es mucho más grande que el tamaño de la memoria caché, la transferencia de esos datos a través del puerto ACP afectará a dicha memoria, por lo que es mejor usar el puerto AFI.

Data Movers (B)

Un *data mover* consta de un componente hardware que transfiere datos entre dos subsistemas (o entre dos componentes del sistema) y una función de la librería específica del sistema operativo seleccionado para generar la aplicación. Su función es transferir datos entre el PS y los aceleradores, y entre los aceleradores. Por ejemplo, un acelerador puede usar un bloque de acceso directo a memoria (DMA) leer y / o escribir en la memoria del procesador a través de los puertos de interfaz del PS.

SDSoC genera distintos tipos de *data movers* según las propiedades y el tamaño de los datos que se transfieren.

- ***Escalar:*** los datos escalares siempre son transferidos por el data mover AXI_LITE.

- ***Array:*** SDSoC puede utilizar diferentes data movers, AXI_DMA_SG, AXI_DMA_SIMPLE, AXI_DMA_2D, AXI_FIFO, dependiendo de los atributos de memoria y el tamaño de los datos del array. Por ejemplo, si el array se asigna mediante malloc () (por lo tanto, la memoria no es físicamente contigua), SDSoC generalmente genera AXI_DMA_SG. Sin embargo, si el tamaño de los datos es inferior a 300 bytes, en su lugar se genera AXI_FIFO, ya que el tiempo de transferencia de datos es inferior a AXI_DMA_SG y ocupa mucho menos recursos de la lógica programable del dispositivo.

  La Tabla 4 muestra la aplicabilidad de estos data movers.

Tabla 4. Selección de data movers.

Data Mover	Physical Memory Contiguity	Data Size (bytes)
AXIDMA_SG	Either	> 300
AXIDMA_Simple	Contiguous	< 8M

AXIDMA_2D	Contiguous	< 8M
AXI_FIFO	Non-contiguous	< 300

Normalmente, el compilador SDSoC analiza el array que se transfiere al acelerador hardware y selecciona el data mover apropiado en consecuencia. Sin embargo, hay situaciones donde tal análisis no es posible. En ese caso, SDSoC emite un mensaje de advertencia y pide que se especifiquen los atributos de memoria a través de pragmas SDS. A continuación se muestra un ejemplo del mensaje:

```
WARNING: [SDSoC 0-0] Unable to determine the memory attributes
```

El pragma para especificar los atributos de memoria es:

```
#pragma SDS data mem_attribute(arg:contiguity|cache)
```

En la sintaxis de esta directiva, el atributo *contiguity* deber ser PHYSICAL_CONTIGUOS o NON_PHYSICAL_CONTIGUOUS (valor por defecto). PHYSICAL_CONTIGUOUS significa que toda la memoria correspondiente al argumento *arg* asociado se asigna usando sds_alloc, mientras que NON_PHYSICAL_CONTIGUOUS significa que toda la memoria se asigna usando malloc o como una variable libre en la pila. El atributo *cache* puede ser CACHEABLE (valor predeterminado) o NON_CACHEABLE. Declarar un array como CACHEABLE significa que el tiempo de ejecución de SDSoC debe mantener la coherencia de caché entre la CPU y el acelerador para la memoria asignada al array.

Este pragma se debe especificar inmediatamente antes de una declaración de función, o inmediatamente antes de otro pragma SDS en la declaración de la función. De esta forma, la directiva se aplica a todas las llamadas a la función.

Si se desea especificar directamente qué data mover usar, se puede emplear el siguiente pragma:

```
#pragma SDS data data_mover(arg:dm)
```

Donde 'dm' puede ser AXIDMA_SG, AXIDMA_SIMPLE, AXIDMA_2D o AXI_FIFO, y arg puede ser un array o un puntero de la función al acelerador. Hay que tener en cuenta que el uso de esta directiva puede hacer que el diseño no funcione en el hardware si no se cumplen los requisitos de la Tabla 4.

- ***Estructura o Clase:*** debido a que una única estructura / clase se aplana, cada miembro de datos usa su propio data mover dependiendo de si es un escalar o un array.

Interfaz del Acelerador (C)

La interfaz del acelerador generada por SDSoC depende también del tipo de dato del argumento o argumentos de la función hardware:

- ***Escalar:*** para un argumento escalar, se genera una interfaz de registro que permite la entrada y salida de datos en el acelerador.

- ***Arrays:*** la interfaz hardware empleada en un acelerador para transferir un array puede ser de tipo RAM o de tipo streaming (transmisión continua), dependiendo de cómo acceda el acelerador a los datos en el array.

  La interfaz RAM permite acceder a los datos de forma aleatoria dentro del acelerador cuando el argumento es un array, siendo necesario transferir todo el array a la memoria del acelerador antes de que se produzca cualquier acceso a dicha memoria. El uso de esta interfaz requiere recursos BRAM en el lado del acelerador para almacenar el array.
  La interfaz de tipo streaming, por otro lado, no requiere memoria para almacenar todo el array. Esto permite al acelerador canalizar el procesamiento de los elementos del array, es decir, el acelerador puede comenzar a procesar un nuevo elemento del array mientras que los anteriores aún se están procesando. No obstante, la interfaz de streaming requiere que el acelerador acceda al array en un orden secuencial estricto y la cantidad de datos transferidos debe ser la misma que espera el acelerador.

  SDSoC, por defecto, generará una interfaz RAM para un array. Sin embargo, el entorno proporciona pragmas SDS que permiten al usuario seleccionar la generación de una interfaz de tipo streaming. El siguiente pragma se puede usar para guiar la generación de una interfaz para el acelerador:

  ```
  #pragma SDS data access_pattern(Argument:access pattern)  // access pattern = SEQUENTIAL | RANDOM
  ```

  El patrón de acceso cuando el argumento es un array es por defecto "RANDOM". Para indicar al compilador sdscc que una función implementada en hardware puede admitir acceso streaming para la transferencia de datos de un array (es decir, a cada elemento se accede una sola vez y en el orden estricto del índice) el argumento debe ser declarado "SEQUENTIAL".

- ***Estructura o Clase:*** cada miembro de la estructura o clase tiene su propia interfaz, dependiendo de si es un escalar o un array.

4.1.3. Arquitecturas de memoria en la implementación hardware de algoritmos de procesado de imágenes y vídeo

La arquitectura de memoria utilizada en la implementación hardware de algoritmos de procesado de imagen y video es crítica para obtener una implementación eficiente. Estos buffers de memoria permiten al usuario tener disponibles los datos temporales y espaciales requeridos por el algoritmo para trabajar en el píxel actual y tienen implicaciones en la latencia y el orden en el que se realizan las operaciones.

Tres de los estilos de buffers de memoria más comunes para los filtros espaciales son los registros de desplazamiento, las ventanas de memoria y los buffers de línea. Los filtros temporales requieren, adicionalmente, el uso de frame-buffers para combinar píxeles de fotogramas sucesivos. Recordemos que ya se discutió la implementación de estos últimos al describir la plataforma hardware base en el Capítulo 3.

Registros de desplazamiento

Los registros de desplazamiento son una de las estructuras de memoria más simples. La idea detrás de un registro de desplazamiento es proporcionar un buffer temporal de datos unidimensional para almacenar los datos entrantes desde una interfaz de transmisión. La duración durante la cual se almacena un dato en el registro de desplazamiento está determinada por la longitud del registro de desplazamiento.

Un ejemplo de un algoritmo típico que se puede implementar con un registro de desplazamiento es un filtro de respuesta de impulso finito (FIR). El diagrama conceptual de un filtro FIR con un registro de desplazamiento se muestra en la Figura 4.3.

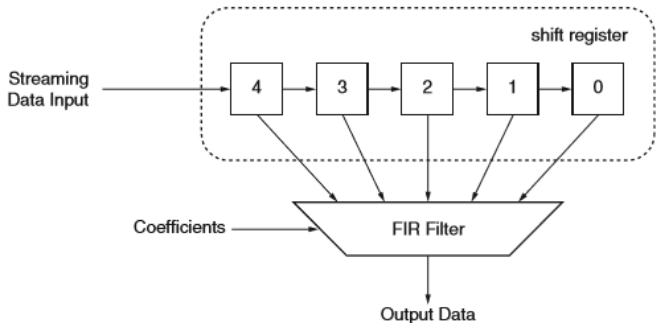


Figura 4.3. *Filtro FIR con registro de desplazamiento.*

Como se puede observar, cada elemento de datos de la interfaz de transmisión entra en el registro de desplazamiento en la posición 4 y es reutilizado cuatro veces más por el filtro FIR antes de ser descartado. Por definición, una interfaz de transmisión de datos

proporciona cada muestra una vez única. Por lo tanto, la muestra de datos se debe almacenar si es necesario utilizarla posteriormente.

Un array es la forma más simple y sencilla de declarar un registro de desplazamiento cuando se utiliza un lenguaje de programación:

```
int A[5];
```

Ventanas de memoria

Una ventana de memoria se define como una vecindad de N píxeles centrados en el píxel P. La ventana de memoria también se puede ver como una colección de registros de desplazamiento, que forma un elemento de almacenamiento de datos bidimensional.

Este tipo de memoria generalmente se implementa como flip-flops debido a las siguientes razones:

- Generalmente no tiene muchos elementos de datos. Solo se almacenan los píxeles necesarios para calcular algunas características de P. Un ejemplo de esto son las ventanas de memoria 3 x 3 utilizadas en la detección de bordes que vamos a implementar como ejemplo.
- Todos los píxeles en el vecindario deben estar disponibles simultáneamente cuando se calcula el valor de P.

La definición más básica de una ventana de memoria en C / C ++ es un array bidimensional. Por ejemplo, una ventana de memoria 3 x 3, B, se puede definir como:

```
int B[3][3];
```

Line Buffer

Un line buffer es un registro de desplazamiento multidimensional que puede almacenar varias líneas de datos de píxeles. Por lo general, los buffers de línea se implementan con los bloques de memoria RAM disponibles en las actuales FPGAs (BRAM) para evitar la latencia de la comunicación con la memoria DDR ubicada fuera del chip. Además, un buffer de línea requiere acceso de lectura y escritura simultáneo, lo que aprovecha al máximo la naturaleza de doble puerto de las BRAM.

Aunque una ventana de memoria es un subconjunto de un buffer de línea, este no se puede usar directamente en la mayoría de los algoritmos de procesado de imágenes y vídeo. El recorrido de la ruta de un píxel entrante al núcleo de cálculo del algoritmo se muestra en la Figura 4.4.

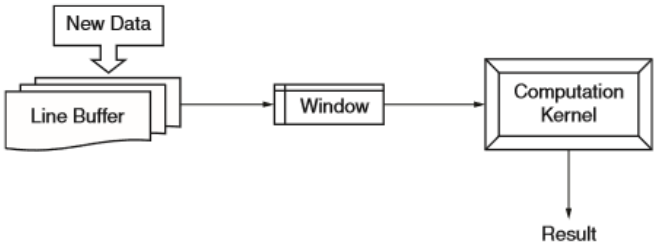


Figura 4.4. *Recorrido conceptual de un píxel en un sistema de procesado.*

Como en el caso de una ventana, un buffer de línea se declara en C / C ++ como una matriz multidimensional. Al determinar la altura del buffer de línea, una regla simple es mantenerlo a la misma altura que la ventana de memoria. Por ejemplo, el buffer de línea correspondiente a la ventana B se declararía como:

```
rgb_pixel D [3] [MAX_COLS]
```

Declarar tres líneas completas de almacenamiento en buffer no es lo más eficiente en términos de uso de recursos FPGA, pero es el modo más fácil de codificar en C / C ++.

En la codificación del algoritmo de procesado de imagen y vídeo, que posteriormente se acelerará en hardware, se implementa un line buffer y una ventana de memoria 3x3 para la adquisición y procesado de los datos del streaming de vídeo de forma eficiente. Para la implementación de estas arquitecturas de memoria se emplean las clases ap_linebuffer y ap_window de la librería HLS de vídeo de Xilinx.

4.2. Algoritmo de procesado: detección de bordes Sobel

El algoritmo de detección de bordes propuesto por Sobel es un algoritmo clásico en el campo del procesamiento de imágenes y video para la extracción de bordes de objetos. El efecto visual tras aplicar una operación Sobel a un frame de vídeo o a una imagen se puede observar en la Figura 4.5.

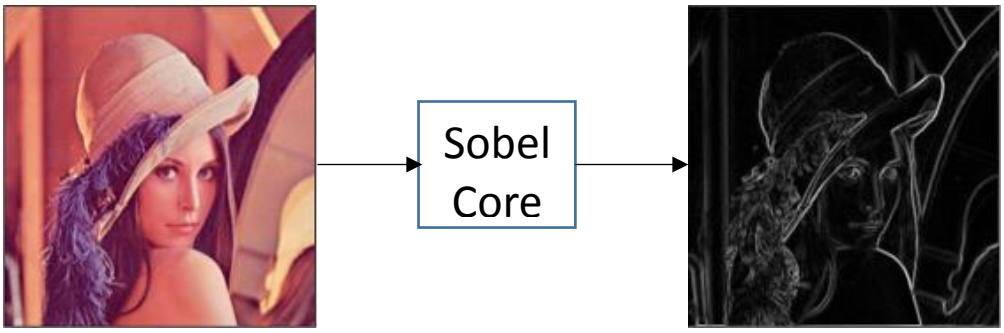


Figura 4.5. *Efecto visual de la detección de bordes usando operadores Sobel.*

La detección de bordes utilizando operadores de Sobel se basa en la premisa de calcular una estimación de la primera derivada de una imagen para extraer información de bordes [2]. Lo que se calcula es la variación de intensidades entre píxeles vecinos (en sentido vertical y horizontal), ya que los bordes de una imagen corresponden a píxeles en los que la variación de intensidad (la derivada estimada) es máxima.

Debido a la carga computacional resultante de calcular las derivadas utilizando operadores de cuadratura y raíz cuadrada, se han adoptado máscaras con coeficientes fijos como una aproximación adecuada para calcular las derivadas en un punto específico. En el caso de la detección de bordes Sobel, las máscaras utilizadas se muestran en la Tabla 5.

Tabla 5. *Máscaras del operador Sobel.*

-1	-2	-1
0	0	0
1	2	1

x

-1	0	1
-2	0	2
-1	0	1

y

A continuación, se muestra el código del algoritmo de detección de bordes Sobel.

```
#include <stdio.h>
#include "ap_video.h"
#include "video_demo.h"

#define ABS(x)          ((x>0)? x : -x)

void DemoSobelFrameHw(u16 srcFrame[DEMO_PIXELS], u16 destFrame[DEMO_PIXELS], u32 fApp)
{
    ap_linebuffer<u8, 3, DEMO_WIDTH> buff_A;
    ap_window<u8,3,3> buff_C;

    int xcoi, ycoi;
    u32 frameWidth = (fApp >> 16);

    for(ycoi = 0; ycoi < DEMO_HEIGHT+1; ycoi++){
        for(xcoi = 0; xcoi < DEMO_WIDTH+1; xcoi++){
#pragma HLS PIPELINE II = 1

            u8 temp, grayIn = 0;
            u16 rIn, gIn, bIn;
            u16 edge;

            if(xcoi < DEMO_WIDTH)
            {
                buff_A.shift_up(xcoi);
                temp = buff_A.getval(0,xcoi);
            }
            if((xcoi < DEMO_WIDTH) & (ycoi < DEMO_HEIGHT))
            {
                pxlInOld = pxlIn;
                pxlIn = srcFrame[ycoi*DEMO_WIDTH+xcoi];
                rIn = ((pxlIn & 0xF800) >> (11-3));
                bIn = ((pxlIn & 0x07C0) >> (6-3));
                gIn = ((pxlIn & 0x003F) << 2);
                grayIn = (rIn * 76 + gIn * 150 + bIn * 29 + 128) >> 8;
                buff_A.insert_bottom(grayIn,xcoi);
```

```
        }

        buff_C.shift_right();

        if(xcoi < DEMO_WIDTH)
        {
            buff_C.insert(buff_A.getval(2,xcoi),0,2);
            buff_C.insert(temp,1,2);
            buff_C.insert(grayIn,2,2);
        }

        if( ycoi <= 1 || xcoi <= 0 || ycoi == DEMO_HEIGHT || xcoi == DEMO_WIDTH)
        {
            edge=0;
        }
        else
        {
            short x_weight = 0, y_weight = 0;
            u8 i, j;
            const short x_op[3][3] = { {-1,0,1},{-2,0,2},{-1,0,1}};
            const short y_op[3][3] = { {1,2,1}, {0,0,0}, {-1,-2,-1}};

            for(i=0; i < 3; i++)
            {
                for(j = 0; j < 3; j++)
                {
                    x_weight = x_weight + (buff_C.getval(i,j) * x_op[i][j]);
                    y_weight = y_weight + (buff_C.getval(i,j) * y_op[i][j]);
                }
            }

            edge = ABS(x_weight) + ABS(y_weight);
            if(edge > 200)
            {
                edge = 255;
            }
            else if(edge < 100)
            {
                edge = 0;
            }

            }

            if(ycoi > 0 && xcoi > 0){
               if (fApp & 0x01)
               {
                  destFrame[(ycoi-1)*DEMO_WIDTH+(xcoi-1)] = pxlInOld;
               }
               else
               {
                  if (xcoi < (frameWidth / 2))
                  {
                     destFrame[(ycoi-1)*DEMO_WIDTH+(xcoi-1)] = pxlInOld;
                  }
                  else
                  {
                     edges norm = (((edge) & 0x00F8) << (11-3)) | (((edge) & 0x00F8) << (6-3)) | (((edge) & 0x00F8) >> 2);

                     destFrame[(ycoi-1)*DEMO_WIDTH+(xcoi-1)] = edge_norm;

                   }
               }
            }
         }
      }
   }
}
```

4.3. Sistema de detección de bordes en vídeo a tiempo real

Como ya se ha mencionado, el sistema se diseña sobre la plataforma base desarrollada en el Capítulo 3. El diagrama de flujo de la aplicación que se utiliza en SDSoC se ilustra en la Figura 4.6.

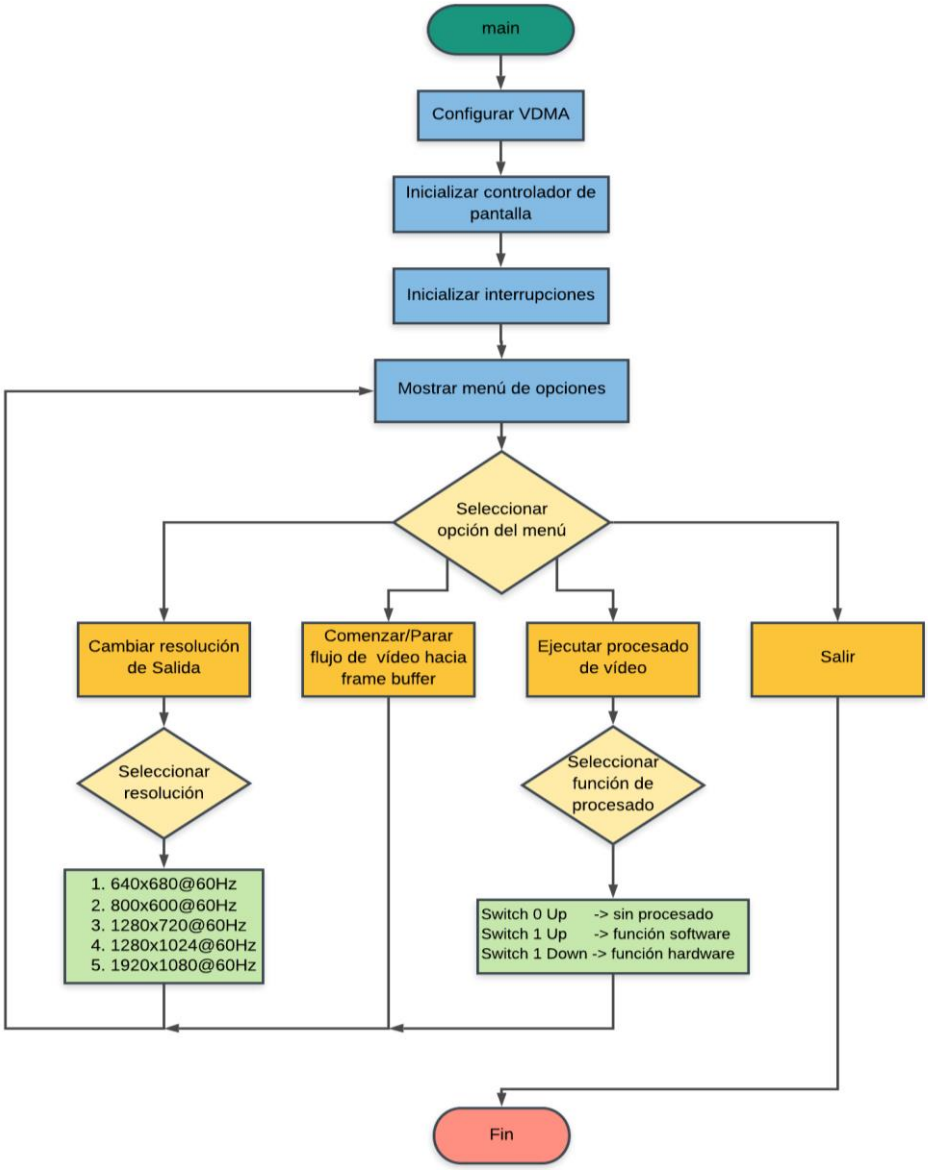


Figura 4.6. *Diagrama de flujo de la aplicación utilizada en SDSoC.*

4.3.1. Aceleración hardware del algoritmo

Para implementar un diseño con aceleración hardware hay que indicarle a SDSoC qué función o funciones van a ser implementadas en la lógica programable y con qué frecuencia de reloj.

La aplicación de procesado de imagen y vídeo que nos ocupa, como se comentó anteriormente, incluye un algoritmo de detección de bordes Sobel que se ha creado como una función aislada dentro de la aplicación. Esta función, denominada "DemoSobelFrameHw", es adecuada para implementarse en hardware ya que realiza un cómputo intensivo de datos y se ha codificado siguiendo las pautas generales que se pueden encontrar en el Anexo II de la memoria.

Una vez creado el proyecto SDSoC y seleccionada la función para ser implementada en hardware, se lleva a cabo la síntesis con HLS. Para ello, se configura el entorno dentro del panel *SDSoC Project Overview* para que implemente la función hardware y la red de movimiento de datos con una frecuencia de reloj 150MHz (frecuencia > 148.5MHz requerida para el procesado en tiempo real, como ya se explicó en el Capítulo 2).

Al finalizar el proceso de síntesis, SDSoC genera de forma automática distintos informes útiles para estimar las prestaciones temporales del acelerador harware o los recursos de hardware que utiliza. Uno de los informes interesantes es el de la red de movimiento de datos, donde se describe la conectividad hardware/software para la función hardware, es decir, qué tipo de data mover y qué puertos del sistema ha seleccionado la herramienta para transferir cada argumento de la función. Los resultados se muestran en la Tabla 6.

Tabla 6. *Red de movimiento de datos generada por SDSoC.*

Accelerator	Argument	IP Port	Direction	Declared Size (bytes)	Pragmas	Connection
DemoSobelFrameHw_0	srcFrame	srcFrame	IN	2073600*2		processing_system7_0_S_AXI_HP2:AXIDMA_SG
	destFrame	destFrame	OUT	2073600*2		processing_system7_0_S_AXI_HP2:AXIDMA_SG

Se puede observar que el entorno ha elegido dos AXIDMA_SG como data mover para cada argumento de la función hardware y utiliza el puerto de alto rendimiento HP2.

El informe también contiene la estimación de los recursos hardware utilizados en la implementación del acelerador. No obstante, esta estimación se puede consultar de forma más clara y resumida en la pestaña principal del informe de estimación SDSoC, tal y como se muestra en la Tabla 7.

Tabla 7. *Estimación de recursos hardware generada por SDSoC.*

Resource utilization estimates for hardware accelerators

Resource	Used	Total	% Utilization
DSP	5	80	6,25
BRAM	9	60	15
LUT	4679	17600	26,59
FF	5969	35200	16,96

Por último, es fundamental analizar la información sobre la latencia y el rendimiento del acelerador sintetizado. La latencia indica el número de ciclos de reloj (del acelerador) que consume una transacción de la función. Esto es, en nuestro caso, el procesado de una imagen completa. El rendimiento mide el número de transacciones por unidad de tiempo. Nos viene indicado indirectamente por el parámetro que HLS denomina Interval y que expresa, también en ciclos de reloj, cada cuánto se comienza una nueva transacción. Cuando una transacción de la función no comienza hasta que no ha terminado la anterior, latencia e Interval están ligados. El interval es la latencia más uno. La Tabla 8 muestra las estimaciones de latencia del acelerador obtenidas en el informe de latencia HLS generado.

Tabla 8. *Informe de latencia y rendimiento del sistema generado por SDSoC.*

+ Latency (clock cycles):
    * Summary:

Latency		Interval		Pipeline
min	max	min	max	Type
12461769	114215218	12461770	114215219	none

    + Detail:
        * Instance:
        N/A

        * Loop:

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- Loop 1	12461768	114215217	11528 ~ 105657	-	-	1081	no
+ Loop 1.1	11526	105655	6 ~ 55	-	-	1921	no
++ Loop 1.1.1	42	42	14	-	-	3	no
+++ Loop 1.1.1.1	12	12	4	-	-	3	no

Se observa que la latencia mínima estimada del acelerador es 12.461.769 ciclos de reloj. Esto supone un problema para el procesado en tiempo real ya que no permitiría procesar 60 frames por segundo como es nuestro objetivo. Para cumplir con esto último, el tiempo de procesado de un frame tiene que estar por debajo de 16,67ms (1/60s). Esto significa que con el reloj que usamos para el acelerador debemos completar el procesado de un frame en menos de 2.500.500 ciclos de reloj (16,67ms*150MHz).

Los pasos a seguir para la creación de un proyecto en SDSoC, la selección de funciones hardware, el proceso de síntesis y la obtención de estimaciones de rendimiento y de la red de movimiento de datos generada se pueden encontrar en el Anexo III de la memoria.

4.3.2. Optimización de la transferencia de datos: directivas SDS

Un sistema hw/sw bien diseñado, generalmente, equilibra el cómputo y la comunicación para que todos los componentes hardware permanezcan ocupados haciendo un trabajo significativo. No obstante, hay muchos factores que afectan el rendimiento general de un sistema híbrido.

Para guiar al compilador en la generación de la interfaz de streaming se usa la directiva mencionada anteriormente, aplicada en la declaración de la función prototipo:

```
#pragma SDS data access_pattern (srcFrame:SEQUENTIAL, destFrame:SEQUENTIAL)
```

Donde srcFrame y destFrame son argumentos tipo array de la función hardware.

En la aplicación de procesado de imagen y vídeo, el software no recibe ni envía datos, se accede a estos a través de un frame-buffer, por lo que no se requiere asegurar la coherencia de caché entre el procesador y el acelerador. Para guiar al compilador en la elección del data mover, se utiliza el siguiente pragma en la declaración de la función hardware, especificándolo inmediatamente antes del pragma introducido para guiar la interfaz del acelerador (pragma data access_pattern):

```
#pragma SDS data access_pattern (srcFrame:PHYSICAL_CONTIGUOUS|NON_CACHEABLE,
                                 destFrame: PHYSICAL_CONTIGUOUS|NON_CACHEABLE)
```

Existe la posibilidad de que el compilador emita el siguiente mensaje de error durante el proceso de síntesis:

```
ERROR: [SDSoC 0:0] The bound callers of accelerator foo have different/ indeterminate data size for port .
```

Este error se debe a que los arrays que contienen los datos de entrada y salida del streaming se pasan a la función implementada en hardware como argumentos de tipo puntero y el compilador no puede inferir el tamaño de los datos a transferir. La siguiente directiva especificará al compilador el tamaño del dato que será transferido.

```
#pragma SDS data copy(p[0:<array_size>])
```

Una vez sintetizado de nuevo el sistema, tras la inclusión de estas directivas, SDSoC genera nuevos informes. El informe de latencia se ha omitido ya que no se ha producido una disminución considerable en los ciclos de reloj, sin embargo, si se ha producido una disminución considerable en la estimación de los recursos hardware que se muestra en la Tabla 9.

***Tabla 9**. Estimación de recursos: optimización de código.*

Resource utilization estimates for hardware accelerators

Resource	Used	Total	% Utilization
DSP	5	80	6,25
BRAM	1	60	1,67
LUT	472	17600	2,68
FF	371	35200	1,05

Comparando los resultados de la Tabla 9 con los obtenidos en Tabla 7 se puede apreciar una disminución considerable en el uso de BRAM, LUTs y FFs. Esto se puede explicar analizando la red de movimientos de datos generada por el entorno, mostrada en la Tabla 10.

***Tabla 10**. Red de movimiento de datos: optimización de código.*

Pragmas	Connection
• mem_attribute:PHYSICAL_CONTIGUOUS.NON_CACHEABLE	processing_system7_0_S_AXI_HP2:AXIDMA_SIMPLE
• mem_attribute:PHYSICAL_CONTIGUOUS.NON_CACHEABLE	processing_system7_0_S_AXI_HP2:AXIDMA_SIMPLE

En este caso, mediante el uso de pragmas SDS, se le ha indicado a SDSoC que los argumentos de la función están asignados en una memoria físicamente contigua. Por lo tanto, la herramienta ha seleccionado dos AXIDMA_SIMPLE ya que es más pequeño y más rápido que los AXIDMA_SG en la transmisión de memoria físicamente contigua, dando lugar a una disminución de los recursos utilizados.

4.3.3. Optimización del acelerador

El algoritmo de procesado implementado en hardware está vinculado a tareas con una elevada carga computacional, por lo que interesa maximizar el rendimiento y minimizar la latencia del acelerador. Aplicando técnicas fundamentales de optimización de HLS se puede aumentar el rendimiento de la función sintetizada en hardware, lo que puede llevar a aumentos significativos en el rendimiento general de la aplicación.

En lenguajes secuenciales como C / C ++, las operaciones en un bucle se ejecutan, por defecto, secuencialmente y la siguiente iteración del bucle solo puede comenzar cuando la última operación en la iteración del bucle actual se completa. La segmentación (pipelining) de bucles permite que las operaciones de un bucle se realicen de manera concurrente en el bloque hardware sintetizado, como se muestra en la Figura 4.7.

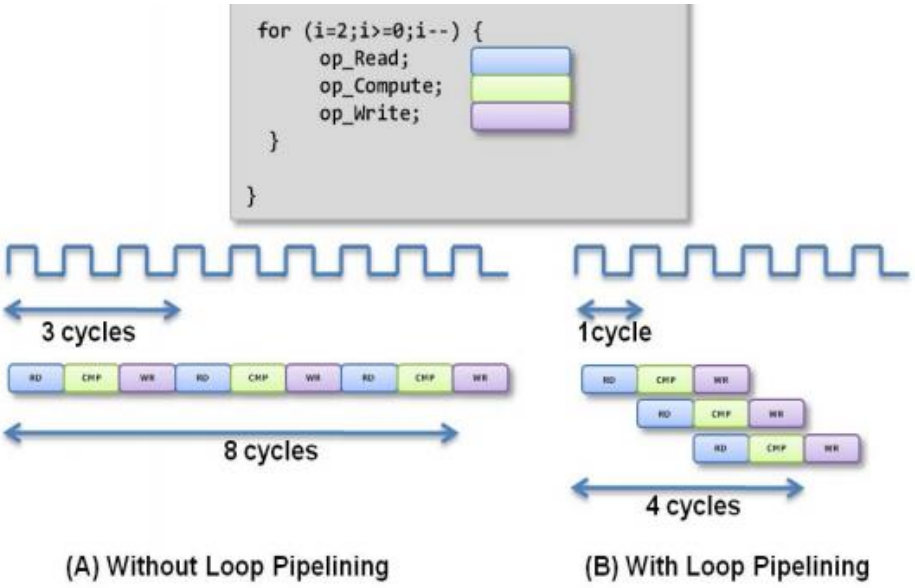


Figura 4.7. *Segmentación (pipelining) de un bucle.*

La segmentación del bucle mejora el rendimiento de la función hardware mediante la explotación del paralelismo entre iteraciones de bucle. La optimización se lleva cabo mediante directivas HLS, siendo 'PIPELINE' la directiva que permite el pipelining en un bucle. Esta directiva o pragma se define del siguiente modo:

```
#pragma HLS PIPELINE II=1
```

Un parámetro importante en la segmentación de bucles es el denominado Intervalo de Iniciación (II), que es el número de ciclos de reloj entre los tiempos de inicio de iteraciones consecutivas de un bucle. En un bucle con pipelining, el intervalo de iniciación (II) es uno si hay un ciclo de reloj entre los tiempos de inicio de dos iteraciones consecutivas.

Hay que tener en cuenta ciertos aspectos a la hora de utilizar la directiva PIPELINE. Cuando tenemos funciones y bucles que contienen sub-bucles o bucles anidados y aplicamos esta directiva, automáticamente se "desenrollan" todos los bucles que se encuentran por debajo en la jerarquía (bucles internos o sub-bucles). Esto puede crear una gran cantidad de lógica, por lo que tiene más sentido segmentar los bucles que se encuentren más abajo en la jerarquía, es decir, los bucles más internos.

En las aplicaciones de imágenes que emplean la técnica de ventana deslizante en la implementación del algoritmo de procesado, como las consideradas en este trabajo, es habitual recorrer el frame de píxeles entrante a través de dos bucles *for* anidados para realizar el tratamiento de datos. Dado los beneficios que supone la segmentación de un bucle y teniendo en cuenta cómo afecta a la generación de la lógica según lo citado en

el párrafo anterior, se utiliza la directiva HLS 'PIPELINE' en el bucle más interno, aprovechando así el paralelismo entre iteraciones del bucle.

Tras aplicar esta directiva y sintetizar de nuevo el sistema, se obtiene el informe HLS que se muestra en la Tabla 11.

Tabla 11. *Informe HLS con acelerador optimizado.*

+ Latency (clock cycles):
    * Summary:

Latency		Interval		Pipeline
min	max	min	max	Type
2076613	2076613	2076614	2076614	none

    + Detail:
        * Instance:
        N/A

        * Loop:

	Latency		Iteration	Initiation Interval		Trip	
Loop Name	min	max	Latency	achieved	target	Count	Pipelined
- Loop 1	2076611	2076611	12	1	1	2076601	yes

Observamos que se ha conseguido el intervalo de iniciación del bucle objetivo. Es decir, cada ciclo de reloj puede comenzar el procesado de un nuevo pixel. De esta forma la latencia se ha reducido considerablemente. El procesado de una imagen consume 2.076.611 ciclos de reloj. Este número es sólo ligeramente superior al número de píxeles de una imagen (2.073.600) pero está por debajo del límite calculado anteriormente para la tasa de frames requerida (2.500.500 ciclos).

Esta reducción en la latencia se debe a que la directiva se aplica en el bucle del algoritmo donde se obtiene y se procesa un dato del array que representa la imagen (es decir, un píxel), por lo que la segmentación de dicho bucle permitirá procesar una muestra por cada ciclo de reloj.

Para ampliar más información sobre directivas HLS se puede consultar [25].

4.3.4. Descripción hardware del sistema sintetizado

Según los análisis realizados en base a los informes, SDSoC ha generado un sistema capaz de procesar en tiempo real, cumpliendo así los objetivos de diseño. Por tanto, es el momento de comprobar las conexiones realizadas por la herramienta sobre la plataforma base inicial. Estas conexiones se ilustran en la Figura 4.8, en la que sólo se muestran las conexiones de interfaz por simplicidad.

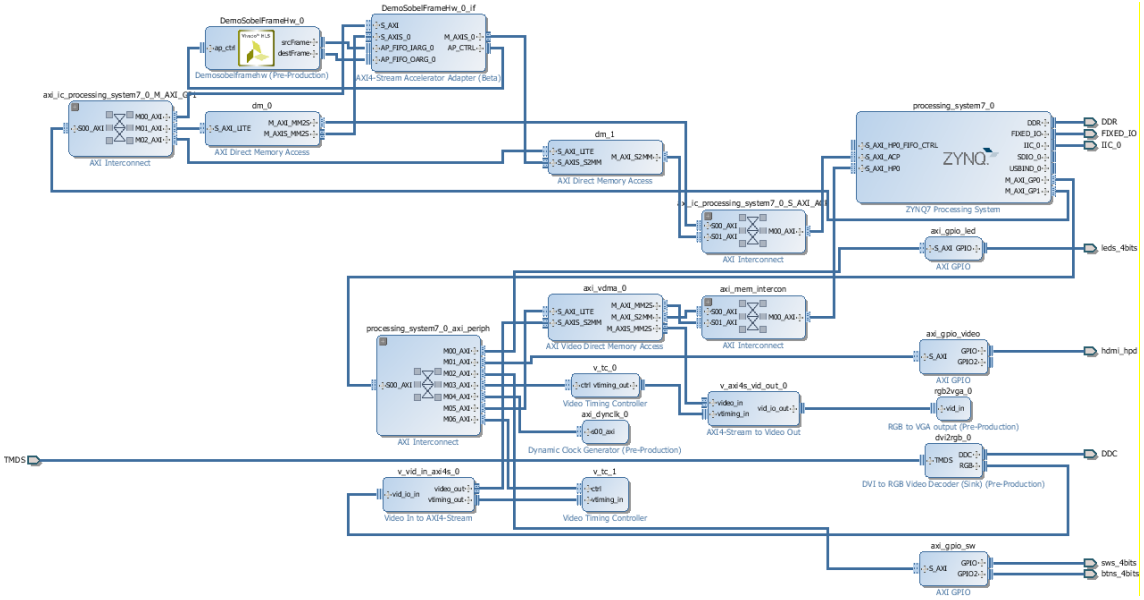


Figura 4.8. *Conexiones realizadas por SDSoC sobre la plataforma base.*

La Figura 4.9 muestra un zoom de los módulos incluidos por SDSoC y las conexiones que ha realizado la herramienta para integrarlos en el sistema**.**

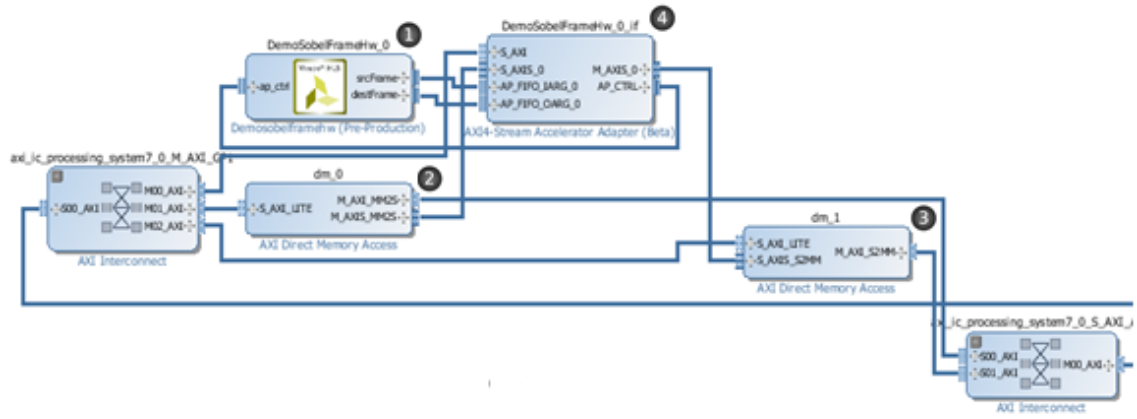


Figura 4.9. *Diagrama de bloques del conjunto del acelerador.*

El bloque del acelerador (1), correspondiente al módulo IP de la función implementada en hardware, recibe datos desde el PS a través de un bloque DMA simple (2). La transmisión de datos hacia el PS se realiza mediante otro bloque DMA Simple (3).

La elección de los data movers por parte de la herramienta es la esperada, ya que en los procesos de optimización se ha guiado a la herramienta en la elección del data mover apropiado para generar una interfaz streaming (4).

El resultado a la salida del sistema de detección de bordes Sobel se ilustra en la Figura 4.10.

b) *Entrada de vídeo.*

a) *Salida de vídeo procesado.*

Figura 4.10. *Procesado de vídeo aplicando operadores Sobel.*

4.4. Otros algoritmos de procesado

Con el fin de demostrar la reusabilidad del sistema, se han implementado otros algoritmos de procesado de imagen y vídeo. Estos algoritmos se describen a continuación.

Laplaciano

El operador de Laplace o Laplaciano se basa en el comportamiento de las segundas derivadas. La detección de bordes empleando operadores Laplacianos se fundamenta en que cuando la imagen presenta un cambio de intensidades a lo largo de una determinada dirección, existirá un máximo en la primera derivada a lo largo dirección, lo que implica un paso por cero en la segunda derivada.

Los píxeles del borde son aquellos tal que el Laplaciano de dos de sus vecinos en posiciones opuestas tienen distinto signo (píxeles de paso por cero). Las máscaras del operador Laplaciano se muestran en la Tabla 12.

Tabla 12. *Máscaras del operador Laplaciano.*

0	1	0
1	−4	1
0	1	0

x

1	1	1
1	−8	1
1	1	1

y

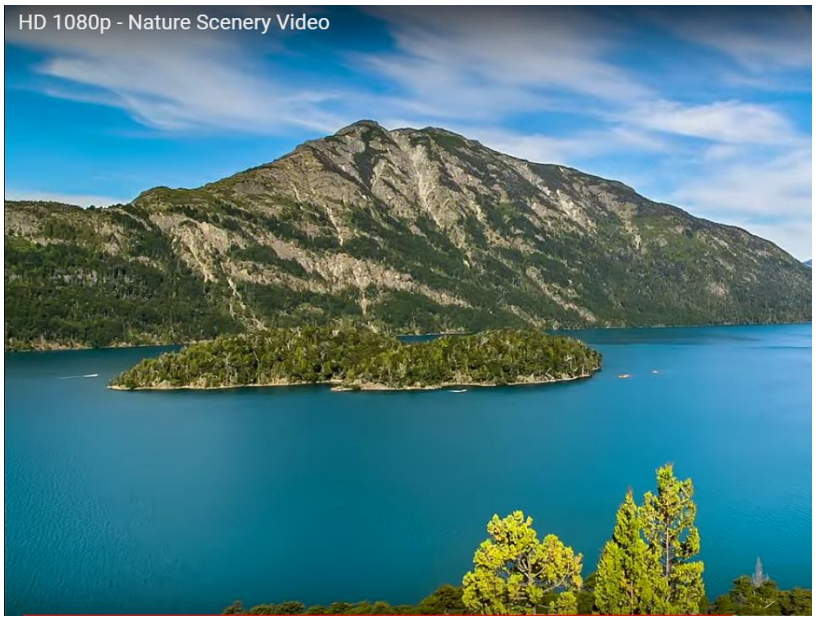
El código de este algoritmo es idéntico al utilizado en la implementación del filtro de detección de bordes Sobel, excepto por las máscaras del operador empleadas. Debido a esto, sólo se mostrará la estimación de recursos utilizados representados en la Tabla 13.

Tabla 13. *Estimación de recursos: detección de bordes con operadores Laplacianos.*

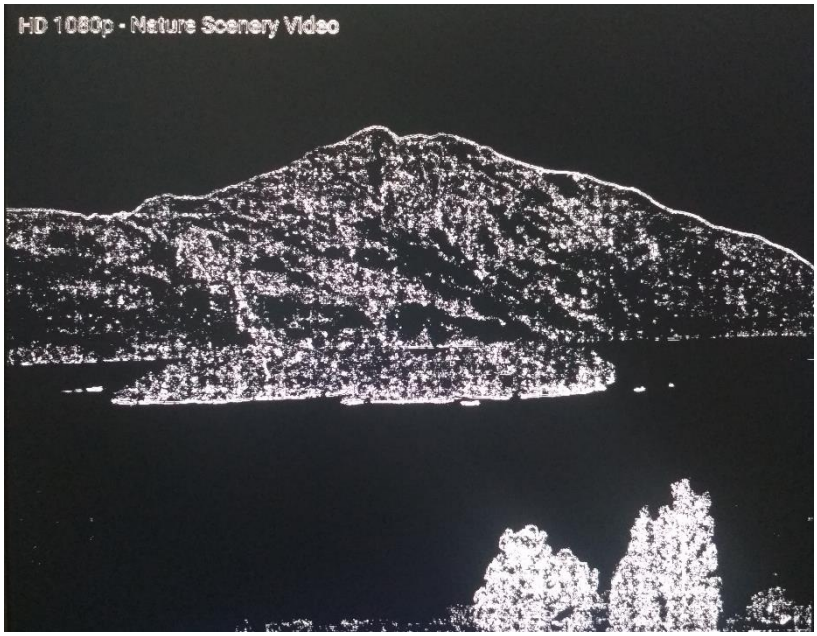
Resource utilization estimates for hardware accelerators

Resource	Used	Total	% Utilization
DSP	3	80	3,75
BRAM	1	60	1,67
LUT	545	17600	3,1
FF	563	35200	1,6

El resultado a la salida del sistema de detección de bordes utilizando operadores Laplacianos se ilustra en la Figura 4.11.



a) *Entrada de vídeo.*



b) *Salida de vídeo procesado.*

Figura 4.11. *Procesado de vídeo aplicando operadores Laplacianos.*

Sepia

Este filtro aplica un color monocromático único de color marrón al flujo de video de entrada. La implementación de este algoritmo no requiere el uso de ventanas de memoria o line buffer. A continuación se muestra el código de este algoritmo.

```
#include <stdio.h>
#include "ap_video.h"
#include "video_demo_sepia.h"

#define ABS(x)          ((x>0)? x : -x)

void DemoSepiaFrameHw(u16 srcFrame[DEMO_PIXELS], u16 destFrame[DEMO_PIXELS], u32 fApp)
{
    int xcoi, ycoi;

    for(ycoi = 0; ycoi < DEMO_HEIGHT+1; ycoi++){
        for(xcoi = 0; xcoi < DEMO_WIDTH+1; xcoi++){
#pragma HLS PIPELINE II = 1
            u8 temp, grayIn = 0;
            u16 rIn, gIn, bIn, rOut, gOut, bOut;
            u16 pxlIn, pxlInOld = 0;

            if((xcoi < DEMO_WIDTH) & (ycoi < DEMO_HEIGHT)){
                pxlInOld = pxlIn;
                pxlIn = srcFrame[ycoi*DEMO_WIDTH+xcoi];

                rIn = ((pxlIn & 0xF800) >> (11-3));
                bIn = ((pxlIn & 0x07C0) >> (6-3));
                gIn = ((pxlIn & 0x003F) << 2);

                grayIn = (rIn * 76 + gIn * 150 + bIn * 29 + 128) >> 8;
                if(grayIn > 255)
                {
                    grayIn=255;
                }
                else if(grayIn < 0)
                {
                    grayIn=0;
                }
                rOut = (grayIn+grayIn *112) >> 8;
                if(rOut> 255)
                {
                     rOut = 255;
                 }
                 else if (rOut < 0)
                 {
                     rOut = 0;
                 }
                 gOut = (grayIn+grayIn *66) >> 8;
                 if(gOut > 255)
                 {
                     gOut = 255;
                 }
                 bOut = (grayIn+grayIn *20) >> 8;
                 if(bOut > 255)
                 {
                     bOut = 255;
                 }
                 else if(bOut < 0)
                 {
                     bOut = 0;
                 }
             }


             if(ycoi > 0 && xcoi > 0){
                 if (fApp & 0x01)
                 {
                     destFrame[(ycoi-1)*DEMO_WIDTH+(xcoi-1)] = pxlInOld;
                 }
                 else
                 {
                     out = (((rOut + 4) & 0x00F8) << (11-3)) | (((bOut + 4) & 0x00F8) << (6-3)) | (((gOut + 4) & 0x00F8) >> 2);
                     destFrame[(ycoi-1)*DEMO_WIDTH+(xcoi-1)] = out
                 }
             }
         }
     }
}
```

Una vez se ha sintetizado, se obtiene la estimación de recursos mostrada en la Tabla 14.

Tabla 14. *Estimación de recursos: filtro Sepia.*

Resource utilization estimates for hardware accelerators

Resource	Used	Total	% Utilization
DSP	6	80	7,5
BRAM	0	60	0
LUT	269	17600	1,53
FF	370	35200	1,05

El resultado a la salida del sistema aplicando el color monocromático característico del filtro sepia se ilustra en la Figura 4.12.

a) *Vídeo de entrada.*

b) *Vídeo de salida procesado.*

Figura 4.12. *Procesado de vídeo aplicando filtro sepia.*

5. Conclusiones y líneas futuras

Las principales conclusiones obtenidas tras la realización de este trabajo se resumen a continuación.

- Se ha explorado y descrito una metodología de codiseño hardware/software sobre dispositivos AP SoCs de Xilinx, centrada en la herramienta SDSoC. Frente a la metodología convencional de diseñar separadamente ambas componentes, esta metodología permite diseñar a partir de una única descripción software. Determinadas partes de este procesado se aceleran en hardware, haciendo uso de la lógica programable del dispositivo.

  Esta metodología requiere el diseño de una plataforma hardware que sirva como base de un proyecto SDSoC y usa internamente una herramienta de síntesis de alto nivel HLS.

- Se ha diseñado una plataforma base para un dispositivo Zynq, siguiendo las directrices necesarias para que sea compatible con el entorno SDSoC. En concreto, esta plataforma está dirigida a la placa ZYBO y permite el procesado de un flujo de vídeo de entrada en formato 1080p@60Hz a través de un puerto HDMI y proporciona la salida mediante un puerto VGA con una resolución máxima de 1080p@60Hz.

- El uso de directivas durante la fase de síntesis de los aceleradores hardware es crítico para obtener resultados competitivos. Se han explorado las distintas opciones de optimización de HLS y se han extraído directrices para guiar a un futuro diseñador.

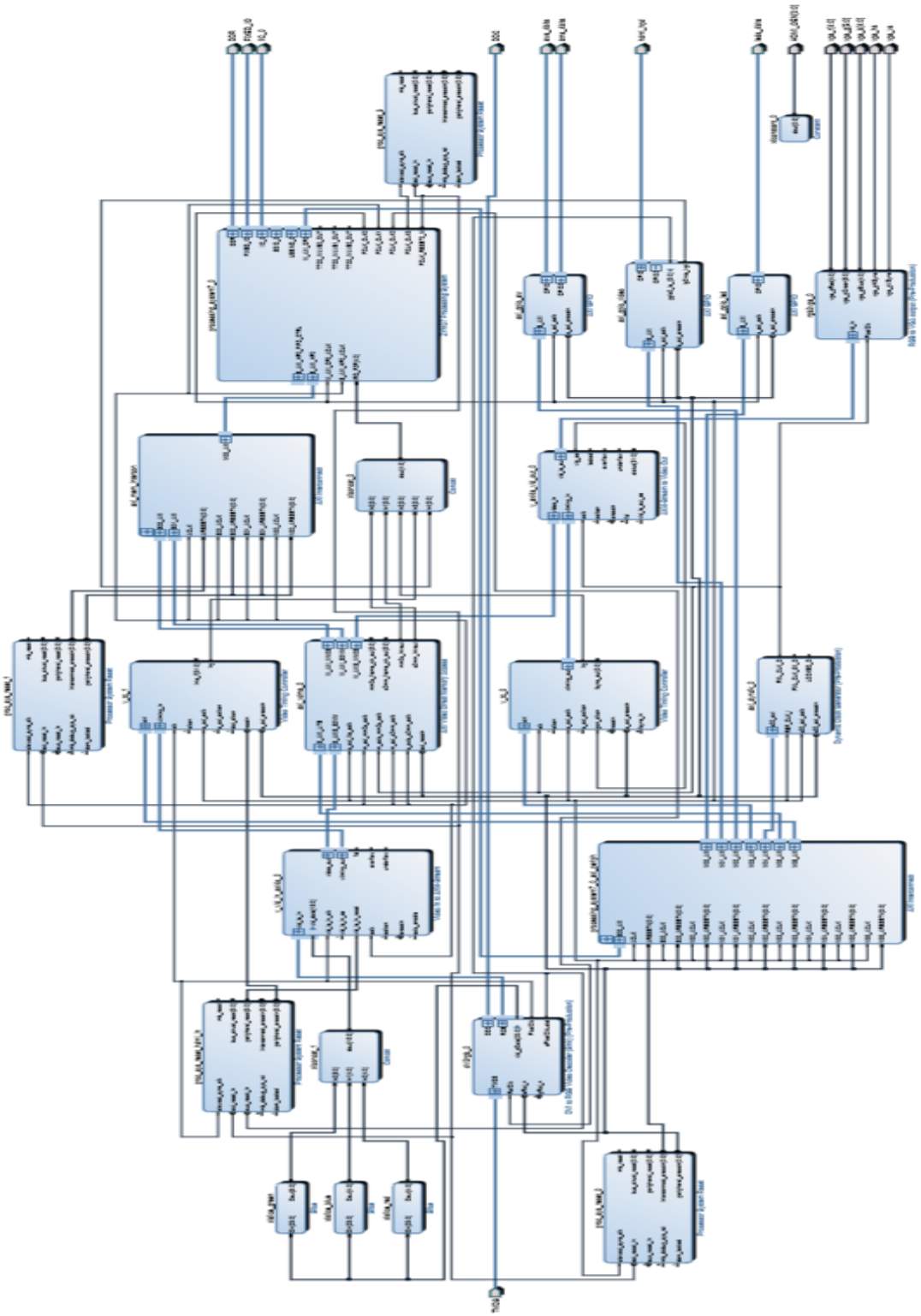
- Se ha desarrollado un sistema de detección de bordes capaz de procesar en tiempo real un vídeo con una resolución de entrada de 1080p. Este sistema se ha generado siguiendo la metodología de codiseño con SDSoC citada anteriormente.

- Se han implementado otros algoritmos de procesado de imagen y vídeo, demostrando así la reusabilidad del diseño y la eficiencia de SDSoC a la hora de prototipar nuevos algoritmos de procesado con aceleración hardware.

- Durante el desarrollo del presente trabajo han surgido una serie de posibles mejoras y líneas futuras de trabajo, las cuales se describen a continuación:
  - Implementación de algoritmos de procesado temporal (p. ej. detección de movimiento).
  - Explorar la implementación de algoritmos que requieran más de una etapa de procesado. Es interesante contemplar tanto la implementación de un pipeline de aceleradores en el que la información pasa de uno a otro como soluciones basadas en aceleradores independientes.
  - Explorar el uso de la librería OpenCV_HLS para desarrollar aplicaciones de procesado de imagen y vídeo con aceleración hardware, siguiendo la metodología de codiseño con SDSoC.

Anexos

A Anexo I: Proyecto Vivado



B Anexo II: Pautas generales para la codificación de aplicaciones en SDSoC

- Las funciones que se implementen en hardware pueden ejecutarse simultáneamente bajo el control de un hilo maestro. Se admiten múltiples hilos maestros.
- Una función hardware de nivel superior (top level) debe ser una función global, no un método de clase, y no puede ser una función sobrecargada.
- No hay soporte para el manejo de excepciones en las funciones hardware.
- Es un error referirse a una variable global dentro de una función de hardware o cualquiera de sus subfunciones cuando otras variables que se ejecutan en el software hacen referencia a esta variable global.
- Las funciones de hardware admiten tipos escalares de hasta 1024 bits, incluyendo double, long long, estructuras empaquetadas, etc.
- Una función de hardware debe tener al menos un argumento.
- Un argumento escalar de salida o de entrada a una función de hardware puede asignarse varias veces, pero solo el último valor escrito se leerá al salir de la función.
- Una función hardware de nivel superior (top level) no debe contener pragmas *HLS interface* en su declaración, ya que el entorno SDSoC las generará de forma automática. No obstante, hay dos directivas de entorno que se pueden especificar para guiar al entorno en la generación del directiva HLS deseada:
  - Para implementar una interfaz de memoria compartida como una interfaz AXI maestro en hardware puede usarse la directiva:

    ```
    #pragma SDS data zero_copy
    ```

  - Para generar una interfaz *streaming* implementada como una interfaz FIFO en hardware se puede especificar la siguiente directiva:

    ```
    #pragma SDS data access_pattern(argument:SEQUENTIAL)
    ```

- Evitar el uso de arrays de tipo *booleano,* ya que presenta un diseño de memoria distinto entre el intérprete ARM y Vivado HLS.
- Evitar usar ap_int <>, ap_fixed <>, hls :: stream, excepto con un ancho de datos de 8, 16, 32 o 64 bits.

C Anexo III: Proyecto SDSoC

Anexo III (A): Creación de un nuevo proyecto

1. Iniciar SDx IDE 2016.4 usando el icono del escritorio o el menú inicio.

2. Una vez iniciado el entorno SDx IDE, aparece el cuadro de diálogo de selección del espacio de trabajo. Clic en el botón ***Browse*** para seleccionar la carpeta en la que se almacenará el proyecto. A continuación, clic en ***OK*** para cerrar el diálogo.

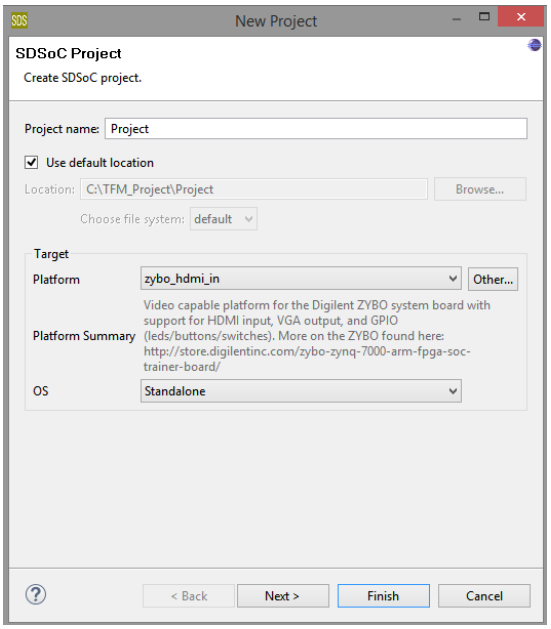
3. Cuando se crea un nuevo espacio de trabajo el entorno de desarrollo se abre automáticamente con la pestaña de bienvenida. Para crear un nuevo proyecto existen dos opciones:

    - Hacer Clic en la opción ***Create SDSoC Project***, accediendo así así cuadro de diálogo New Project

    - Cerrar la pestaña de bienvenida, permitiendo el acceso a la barra de menú del IDE. A continuación, abriremos el cuadro de diálogo New Project seleccionando ***File -> New -> Xilinx SDSoC Project***

4. Especificar el nombre del proyecto en el campo *Project name* (por ejemplo Project) y hacer clic en **Next**.

5. En la sección **Target**, clic **Other...** en el apartado *Platform* para seleccionar la plataforma dentro del directorio en el que se encuentre almacenada. A continuación, seleccionar Standalone en el apartado *OS (Operating System)*. el cuadro de diálogo New Project debe quedar como se muestra en la imagen.

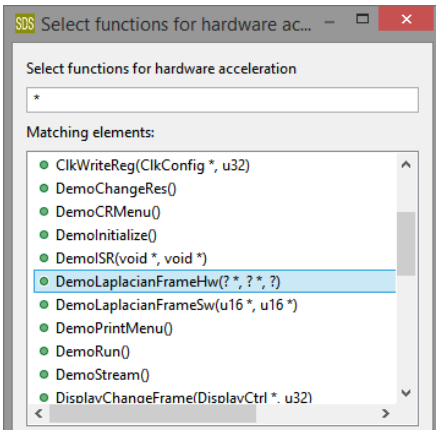


6. Finalmente, hacer clic en **Finish**.

Anexo III (B): Añadir funciones para implementación hardware

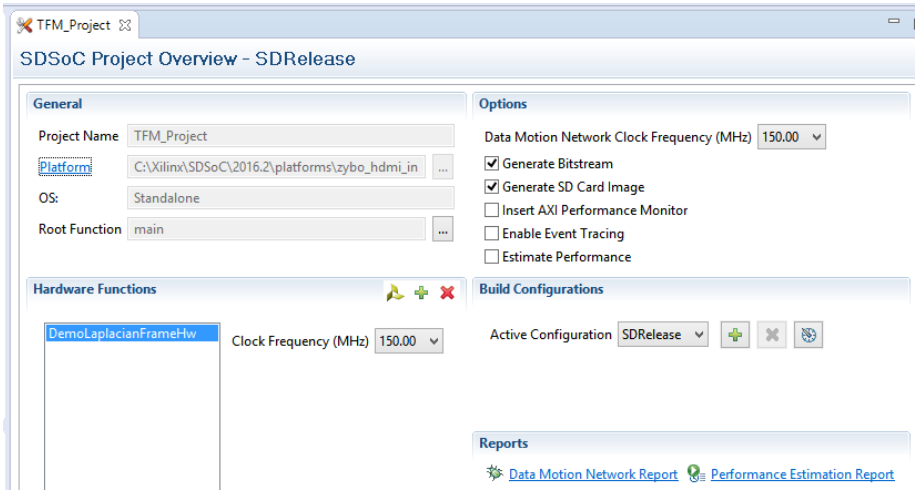
Como se comentó en el Capítulo 4, la aplicación cuenta con una función adecuada para ser implementada en hardware (denominada DemoSobelFrameHw). Para indicarle al entorno qué función será añadida en hardware debemos seguir los siguientes pasos:

1. En la pestaña etiquetada como TFM_Project, en el panel HW functions de SDSoc Project Overview, clic en el botón ***ADD hardware Functions...*** para abrir el cuadro de diálogo de especificación de funciones hardware.



   Si la pestaña TFM_Project no está visible, basta con hacer doble clic en el archivo project.sdx de la pestaña *Project Explorer*.

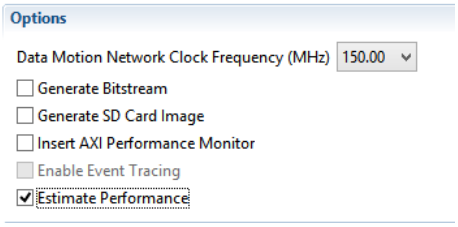
2. En el apartado Hardware Functions de la pestaña *SDSoC Project Overview* seleccionar la frecuencia de reloj de la función (o funciones) hardware. A continuación, seleccionar la misma para el reloj de la red de movimiento de datos, en el apartado *Options*.



Anexo III (C): Síntesis, Estimación del rendimiento y Red de Movimiento de Datos

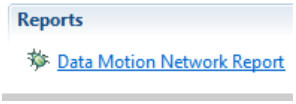
Una vez creado el proyecto y añadida la función o funciones a implementar en hardware, el siguiente paso es compilar el proyecto. Una compilación en la que se genera un bitstream completo puede llevar más tiempo que una compilación de software, por lo que SDSoC proporciona opciones de estimación de rendimiento para calcular la estimación de la aceleración.

Estimar la aceleración es un proceso de dos fases. Primero, el entorno SDSoC compila las funciones de hardware y genera el sistema. En lugar de sintetizar el sistema y generar un bitstream, el compilador calcula una estimación del rendimiento en función de las latencias estimadas para las funciones hardware y las estimaciones de tiempo de transferencia de datos para las llamadas a las funciones de hardware. Para realizar las estimaciones sin generar un bitstream, se debe marcar únicamente la casilla Estimate Performance en el apartado *Options.*



Para generar un sistema completo (incluyendo bitstream), hay que marcar las casillas *Generate Bitstream* y *Generate SD Card Image.* A continuación, pulsamos el botón *Build* ubicado en la barra de herramientas.

Una vez finalizada la síntesis, se puede consultar la red de movimiento de datos generada por SDSoC haciendo clic en *Data Motion Network Report* del apartado *Reports.*



Se puede consultar la Guía de usuario del entorno SDSoC: Introducción al entorno SDSoC [35] para obtener más información.

Bibliografía

[1] D. Bailey, Design for Embedded Image Processing on FPGAs, Wiley-IEEE Press, 2011.

[2] R. C. G. a. R. E. Woods, Digital Image Processing, 3rd Edition, Pearson, 1998.

[3] A. K. R. Tinku Acharya, Image Processing: Principles and Applications, John Wiley & Sons, 2005.

[4] I. Xilinx, Video Timing Controller,PG016, 2015.

[5] I. Xillinx, AXI4-Stream Video IP and System Design Guide, UG934, 2016.

[6] Digilent, ZYBO Reference Manual, 2014.

[7] R. A. E. M. A. E. D. N. Louise H. Crockett, The Zynq Book, 2015.

[8] ARM, AMBA AXI and ACE Protocol Specification: AXI3, AXI4, and AXI-Lite, ACE and ACE-Lite, 2013.

[9] I. Xilinx, LogiCORE IP MicroBlaze Micro Controller System, Product Specification, DS865, 2012.

[10] I. Xilinx, Embedded Processor Hardware Design,UG898, 2016.

[11] I. Xilinx, Zynq-7000 Technical Reference Manual, UG585, 2014.

[12] I. Xilinx, 7 Series DSP48E1 Slice User Guide",UG479, 2014.

[13] I. Xilinx, 7 Series FPGAs Memory Resources User Guide, 2014.

[14] R. Wilson, «"Truth About Xilinx Love Affair with AMBA",» 2010.

[15] I. Xilinx, "Xilinx and ARM Announce Development Collaboration", press release, 2009.

[16] ARM, AMBA 4 AXI4-Stream Protocol Specification, 2010.

[17] I. Xilinx, Vivado Design Suite User Guide: System-Level Design Entry, UG895, 2016.

[18] I. Xilinx, Vivado Design Suite User Guide: Designing with IP, UG896, 2016.

[19] I. Xilinx, Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator, UG994, 2016.

[20] I. Xilinx, Vivado Design Suite User Guide: Logic Simulation, UG900, 2016.

[21] I. Xilinx, Vivado Design Suite User Guide: Synthesis , UG901, 2016.

[22] I. Xilinx, Vivado Design Suite User Guide: Implementation, UG904, 2016.

[23] I. Xilinx, Vivado Design Suite User Guide: Programming and Debugging, UG908, 2016.

[24] I. Xilinx, Vivado Design Suite User Guide: Using the Vivado IDE, UG893, 2016.

[25] I. Xilinx, SDSoC Environment User Guide UG1027, 2016.

[26] T. L. a. D. W. Neuendorffer, Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries", Xilinx Application Note, XAPP1167, 2013.

[27] Digilent, RGB-to-DVI IP Core User Guide, 2016.

[28] I. Xilinx, AXI Video Direct, November 2016.

[29] X. Inc, LogiCORE IP AXI Interconnect, 2013.

[30] I. Xilinx, LogiCORE IP Processing System 7, 2013.

[31] Digilent, RGB-to-VGA 1.0 IP Core User Guide, 2015.

[32] I. Xilinx, Processor System Reset Module, 2015.

[33] I. Xilinx, LogiCORE IP Concat, 2016.

[34] I. Xilinx, SDSoC Environment Platforms and Libraries,UG1146, 2016.

[35] I. Xilinx, SDSoC Environment Getting Started, UG1028, 2016.

[36] I. Xilinx, LogiCORE IP, April 2014.

[37] C. Kohn, Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All Programmable SoC Devices", Xilinx Application Note, XAPP1159, 2013.

[38] M. Mitchell, Zynq for Video Applications, Xilinx presentation, 2012.

[39] F. M. Vallina, «Implementing Memory Structures for Video Processing in the Vivado HLS Tool, Xilinx Application Note, XAPP1167, v2.0,» 2012.

[40] I. T. Union, “H265: High Efficiency Video Coding”, Recommendation H.265(04/13), 2013.

[41] I. Xilinx, LogiCORE IP Slice, 2016.

[42] I. Xilinx, Zynq-7000 All Programmable SoC Software Developers Guide, UG821, 2015.

[43] I. Xilinx, LogiCORE IP Video, 2013.

[44] I. Xilinx, LogiCORE IP Video In, 2014.

[45] I. Xilinx, AXI4 IP Catalogue.

[46] ARM, Cortex-A9 NEON Media Processing Engine Technical Reference Manual, 2011.

Índice de figuras

Índice de tablas