UNIVERSIDAD AUTONOMA DE MADRID

Escuela Politécnica Superior

20|21

Trabajo fin de grado

Diseño FPGA desde alto nivel usando Python y Zynq aplicado al procesamiento de Video

Santiago Nazar Alaez

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

Campus Internacional excelencia UAM CSIC+

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

TRABAJO FIN DE GRADO

Diseño FPGA desde alto nivel usando Python y Zynq aplicado al procesamiento de Video

Autor: Santiago Nazar Alaez
Tutor: Gustavo Sutter

junio 2021

Todos los derechos reservados.
Queda prohibida, salvo excepción prevista en la Ley, cualquier
forma de reproducción, distribución comunicación pública
y transformación de esta obra sin contar con la autorización
de los titulares de la propiedad intelectual.
La infracción de los derechos mencionados puede ser constitutiva de
delito contra la propiedad intelectual *(arts. 270 y sgts. del Código Penal).*

DERECHOS RESERVADOS
© 4 de Mayo de 2021 por UNIVERSIDAD AUTÓNOMA DE MADRID
Francisco Tomás y Valiente, n^o 1
Madrid, 28049
Spain

Santiago Nazar Alaez
Diseño FPGA desde alto nivel usando Python y Zynq aplicado al procesamiento de Video

Santiago Nazar Alaez
C\ Corbeta Nº 14

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Agradecimientos

Me gustaría agradecer en la realización de este trabajo y en general de toda mi carrera universitaria a mi familia, que siempre han estado ahí y han sido mi apoyo y mi ayuda a lo largo de mi formación a lo largo de toda mi vida, a mi novia, que me ha motivado a llegar hasta donde he llegado y que ha estado junto a mí en todo momento, sobre todo en aquellos en los que todo se complicaba.

Seguir agradeciendo a Gustavo, mi tutor de este trabajo de fin de grado, por haberme motivado y acompañado con él durante todo el curso y por el tiempo dedicado a lo largo de todo su desarrollo. También a la Escuela Politécnica Superior y a la Universidad Autónoma de Madrid por haberme brindado la oportunidad de realizar esta increíble carrera.

Por último a mis amigos y compañeros de la universidad con los que he pasado tanto tiempo y he compartido tantas horas dentro y fuera de las aulas de esta escuela.

Resumen

A día de hoy puede llegar a resultar complicado encontrar a alguien con el conocimiento y la experiencia suficiente como para desarrollar tanto software como hardware al mismo tiempo, mucho más teniendo en cuenta la complejidad de las herramientas de desarrollo hardware, esta es la principal razón por la que Xilinx creó este proyecto llamado PYNQ.

Teniendo esto en cuenta, se va a diseñar este trabajo con el objetivo de estudiar, desde un punto de vista eminentemente práctico, todo el flujo de diseño del proyecto PYNQ, se analizarán especialmente sus objetivos, eficiencia, versatilidad y funcionalidades.

El elemento central de este trabajo consistirá en un ejemplo de implementación de este mismo proyecto mediante una pequeña alteración hardware y un análisis de todos los pasos seguidos para su correcta implementación y uso. A este paso le será añadida una completa evaluación de los resultados obtenidos y de la eficiencia del proyecto desde el punto de vista de un diseñador.

Palabras clave

PYNQ, ZCU104, Trabajo fin de grado, Xilinx, Zynq, MPSoC, Overlays, Python, Jupyter, codiseño hardware-software

Abstract

Nowadays, it can be difficult to find someone with enough knowledge and experience to develop both software and hardware at the same time, especially taking into account the complexity of hardware development tools, this is the main reason why Xilinx created this project called PYNQ.

With this in mind, this work will be designed with the aim of studying, from an eminently practical point of view, the whole design flow of the PYNQ project, especially its objectives, efficiency, versatility and functionalities will be analysed.

The central element of this work will consist of an example of implementation of this same project by means of a small hardware alteration and an analysis of all the steps followed for its correct implementation and use. To this step will be added a complete evaluation of the results obtained and the efficiency of the project from a designer's point of view.

Keywords

PYNQ, ZCU104, Final Degree Project, Xilinx, Zynq, MPSoC, Overlays, Python, Jupyter, hardware-software co-design

ÍNDICE

1 INTRODUCCIÓN **1**

1.1 Motivación 1

1.2 Objetivos 1

1.3 Organización de la memoria 2

2 ESTADO DEL ARTE **3**

2.1 SoC reconfigurables 3

2.1.1 Flujo de diseño hardware 5

2.1.2 Flujo de diseño software 6

2.2 El proyecto PYNQ 6

2.2.1 Overlays de PYNQ 6

2.2.2 Entorno de Python 7

2.2.3 Notebooks de Jupyter 8

2.3 La plataforma ZCU104 9

2.4 Uso de HLS para diseño hardware 10

2.4.1 Diseño hardware 11

2.4.2 Hardware Description Language 11

2.4.3 High-Level Synthesis 12

3 DISEÑO DE REFERENCIA **13**

3.1 Diseño de bloques 13

3.2 Diseño del bloque de video 15

3.3 Diseño de bloques del HDMI de salida 16

4 IMPLEMENTACIÓN DEL PROYECTO **17**

4.1 Configuración de la ZCU104 17

4.2 Diseño inicial del IP core 18

4.3 Modificación del overlay hardware 19

5 EVALUACIÓN Y RESULTADOS **25**

5.1 Pruebas 25

5.1.1 Pruebas realizadas sobre el código del IP core 25

5.1.2 Pruebas realizadas sobre el diseño final en la placa 31

5.2 Evaluación de prestaciones 34

5.3 Esfuerzo de diseño 36

6 CONCLUSIONES **39**

6.1 Trabajos futuros 40

Bibliografía **41**

Definiciones **43**

Acrónimos **45**

Apéndices **47**

A Desarrollo y pruebas del IP core **49**

A.0.1 Diseño IP core 49

A.0.2 Pruebas IP core 55

Listas

Lista de algoritmos

Lista de códigos

4.1 Funcion filtro. 18

5.1 Declaraciones testbench. 26
5.2 Main del testbench. 27
5.3 Main del testbench. 28
5.4 Posibles casos parámetro. 29

A.1 Funcion grises declaraciones. 50
A.2 Funcion grises contenido. 51
A.3 Funcion filtro declaraciones. 51
A.4 Funcion filtro switch parte 1. 52
A.5 Funcion filtro switch parte 2. 53
A.6 Funcion filtro salida. 54
A.7 Funcion image_to_stream. 55
A.8 Funcion stream_to_image. 56

Lista de cuadros

Lista de ecuaciones

Lista de figuras

2.1 Diagrama bloques SoC 4
2.2 Flujo diseño embebido 5
2.3 Flujo hardware 5
2.4 Imagen bloques pynq 7
2.5 Resumen PYNQ 9

2.6 Imagen placa a utilizar 10
2.7 Flujo de diseño HLS 12

3.1 Imagen diseño bloques completo ZCU104 14
3.2 Imagen diseño bloques completo 15
3.3 Imagen diseño bloques video 15
3.4 Imagen diseño bloques hdmiout 16

4.1 Imagen registros offset 20
4.2 Imagen añadir IP 20
4.3 Imagen block design hdmi out 21
4.4 Imagen generacion bitstream 21
4.5 Imagen exportacion block design 22
4.6 Imagen placa encendida 23
4.7 Imagen carpeta filter 23
4.8 Imagen inicio jupyter 24
4.9 Imagen nuevo notebook 24
4.10 Imagen carga overlay 24

5.1 Figuras de prueba 0, 1, 2, y 3 29
5.2 Figuras de prueba 4, 5, 6, 7 y else 30
5.3 Imagen codigo python inicializacion hdmis 31
5.4 Imagen pantalla de salida configuracion hdmis inicial 32
5.5 Imagen codigo prueba python 32
5.6 Figuras de prueba final 0, 3, 5, 6 y 7 33
5.7 Imagen utilización proyecto 35
5.8 Imagen tiempos proyecto 36
5.9 Imagen relojes proyecto 36
5.10 Diagrama de Gantt 37

Lista de tablas

Lista de cuadros

1 INTRODUCCIÓN

Python Productivity for Zynq (PYNQ) es un proyecto de código abierto diseñado por la compañía tecnológica estadounidense Xilinx que facilita el uso de sus plataformas. Utilizando el lenguaje y las bibliotecas de Python los diseñadores pueden aprovechar las ventajas de la lógica programable y los microprocesadores para construir sistemas electrónicos más capaces, eficaces y prácticos. PYNQ puede utilizarse con Zynq, Zynq UltraScale+, Zynq Radio Frequency Simulation Operations Center (RFSoC) , placas aceleradoras Alveo y AWS-F1 para crear aplicaciones de alto rendimiento.

1.1. Motivación

En esta memoria del Trabajo de fin de grado (TFG) se busca analizar en profundidad los objetivos y las utilidades del proyecto de PYNQ , se mostrará como es capaz de permitir, a cualquier persona con mínimos conocimientos acerca de diseño hardware, alterar este mismo en una de sus plataformas de una forma simple, rápida y efectiva, pudiendo posteriormente controlarlo mediante el lenguaje de programación de alto nivel Python, desde un Notebook de Jupyter utilizando tan solo conocimientos software.

Así mismo se observará la sencillez de su uso junto a todo su flujo de diseño, al cual le debemos esta sencillez a la hora de modificarlo y de darle uso conforme a los objetivos buscados.

1.2. Objetivos

El objetivo principal de este trabajo es mostrar, de forma práctica, como mediante un ejemplo, podemos realizar una alteración hardware en una de las placas de desarrollo basada en Zynq UltraScale+ (MPSoC) y provistas por Xilinx.

Para ello se añadirá un pequeño módulo hardware al proyecto maestro de la placa, del cual se tratará más adelante, y que tendrá un parámetro que se podrá controlar mediante Python desde un Notebook de Jupyter, las funcionalidades que se le han dado a este módulo son de filtrado de color en

una imagen o video saliente por los puertos High-Definition Multimedia Interface (HDMI) de la placa en una pantalla, pudiéndose alterar la gama de colores en que se muestra el video tan solo alterándose el parámetro en el código software.

Con esto se busca demostrar como, tan solo habiendo hecho cambios mínimos en el diseño hardware y sin necesidad de un conocimiento profundo acerca de este mismo hardware y de sus herramientas de diseño, se ha podido modificar el mismo llegando a poder controlarlo enteramente mediante programación de alto nivel.

Otro de nuestros objetivos se remite a realizar un análisis profundo acerca de la potencialidad, y posibles funcionalidades, de este prometedor proyecto en la actualidad, así como su desarrollo y el uso que se le está dando ahora mismo.

1.3. Organización de la memoria

Esta memoria se ha estructurado de tal manera que está organizada por capítulos, estos capítulos ya visibles en el índice, constan de cierto número de apartados y sub apartados en función de la importancia que tienen dentro de este trabajo, estos capítulos se resumen a continuación.

ESTADO DEL ARTE En este capítulo se va a tocar todos los temas relacionados con el entorno del proyecto elegido y es donde se encuentra todo el conocimiento previo necesario para comprender y llevar a cabo el proyecto a desarrollar en capítulos posteriores, se va a ver en profundidad los System on a Chip (SoC) reconfigurables y su flujo de diseño, el proyecto PYNQ en más detalle, la plataforma sobre la que se va a trabajar y a implementar el proyecto y el uso de síntesis de alto nivel utilizado para el diseño hardware.

DISEÑO DE REFERENCIA Aquí se explicarán los pasos que se han dado para entender el diseño maestro, el diseño de referencia, a la vez que todas las interconexiones que hay entre los elementos que lo conforman.

IMPLEMENTACIÓN DEL PROYECTO En este capítulo consiste se realizará la modificación del overlay que hemos realizado a modo de ejemplo.

EVALUACIÓN Y RESULTADOS Aquí se recogerán los resultados hardware así como la evaluación de prestaciones y el esfuerzo dedicado como diseñador en esta demostración.

CONCLUSIONES En este último capítulo se mostrarán las conclusiones alcanzadas tras terminar todo el proceso de implementación y evaluación del trabajo.

FUTURO TRABAJO Actualmente dentro de la empresa Xilinx se están dedicando recursos y esfuerzos a potenciar, mejorar y comercializar este proyecto debido a la enorme practicidad de este mismo, por lo que en este apartado trataremos sobre ello.

2

ESTADO DEL ARTE

En este capítulo se va a tratar con todos los conocimientos previos adquiridos, y necesarios, para llevar a cabo el proyecto, y por tanto este TFG . Esto es, el entorno sobre el cual se ha desarrollado y el papel que cumple hoy en día la tecnología PYNQ en el mundo actual.

2.1. SoC reconfigurables

Se va a comenzar hablando de los SoC , para comprender qué es un SoC nos hemos guiado por el artículo citado en [1], en él se verá que estos consisten en la integración de los elementos o módulos que componen un sistema electrónico en un único circuito integrado o chip, de ahí su nombre, su origen se remonta al auge de ciertas grandes compañías especializadas en la electrónica y la fabricación de móviles, su diseño más básico se puede apreciar en la figura 2.1. El principal objetivo de estas compañías electrónicas de entonces era minimizar el tamaño de cualquier componente electrónico que fabricasen, buscando sobre todo, hacer móviles más pequeños y manejables. De ahí nacieron los SoC , un SoC es un chip el cual se diferencia de los demás chips en que está formado por una gran cantidad de componentes diferentes los cuales son los que le dan su nombre, cada uno de ellos tiene una función distinta, pero entre ellos tienen funcionalidades combinadas.

Habiendo estudiado la definición de SoC se va a intentar entender que quiere decir reconfigurable. Para ello, y siguiendo el artículo de [3], se verá que los SoC reconfigurables proporcionan el mismo tipo de soporte de Internet Protocol (IP) personalizada, excepto que la IP se implementa utilizando una matriz reconfigurable. El software debe configurar el hardware antes de poder utilizarlo. Pero a partir de ahí, el software del SoC basado en la plataforma y el software del SoC reconfigurable serán muy similares, asumiendo que el núcleo del microprocesador es el mismo o similar y la funcionalidad de la IP tiene las mismas características.

El hecho de que los SoC reconfigurables estén disponibles en el mercado, no significa que no sean necesarias las herramientas de codiseño y verificación. De hecho, son fundamentales para aprovechar las capacidades de reconfiguración de los chips. La empresa Atmel ofrece herramientas de última generación que permiten el desarrollo simultáneo de hardware y software.

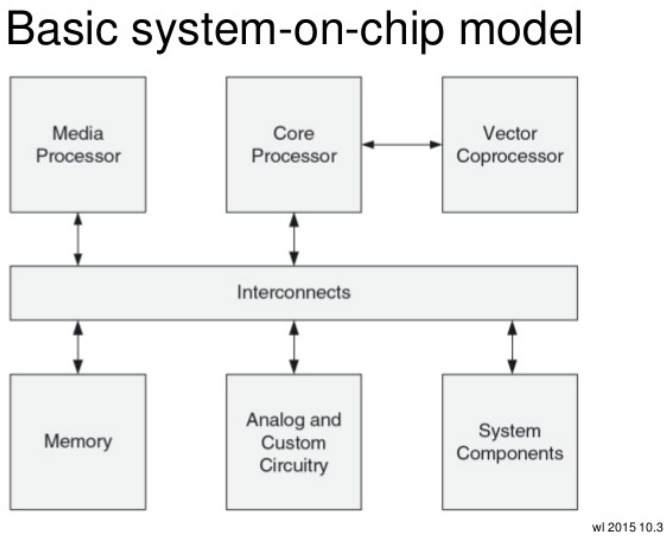


Figura 2.1: Diagrama de bloques de un SoC básico. Extraida de [2]

Los diseños de SoC reconfigurables tienen dos características que los diferencian de otros diseños de SoC . La primera es la posibilidad de cambiar la funcionalidad del hardware simplemente alterando el código que realiza la inicialización del sistema.

El segundo punto es una derivación de esto. La reconfiguración puede producirse un número arbitrario de veces tras la inicialización del hardware. Varias aplicaciones pueden utilizar esta característica. Por ejemplo, un reproductor multimedia podría configurar un códec de hardware para el tipo de flujo de datos que se utiliza, como por ejemplo Moving Picture Experts Group 1 (MPEG-1) o Moving Picture Experts Group 2 (MPEG-2) . Un modo similar sería utilizado por un cliente o servidores privados, Virtual Private Network (VPN) que puede utilizar diferentes métodos de cifrado implementados en el hardware.

Los sistemas integrados son complejos. Las partes de hardware y software de un diseño integrado son proyectos en sí mismos. Fusionar los dos componentes del diseño para que funcionen como un solo sistema crea retos adicionales. Si se añade un proyecto de diseño de Field Programmable Gate Array (FPGA) , la situación puede volverse muy complicada. Para simplificar el proceso de diseño, Xilinx proporciona varios conjuntos de herramientas con los que es necesario familiarizarse y que siguen un flujo de desarrollo tal y como vemos en la figura 2.2.

Estas herramientas son el Integrated Development Environment (IDE) de Vivado, el cual incluye la herramienta IP integrator, que se puede utilizar para unir un diseño basado en procesador. Esta herramienta, combinada con la plataforma de software Xilinx Vitis, proporciona un entorno integrado para diseñar y depurar sistemas basados en microprocesadores y aplicaciones de software embebido.

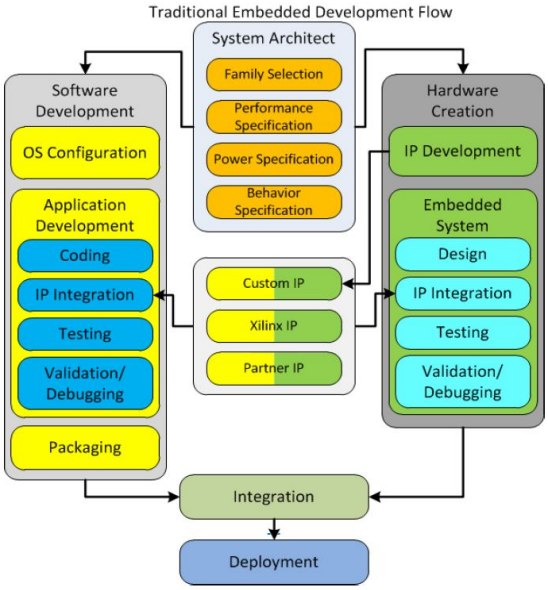


Figura 2.2: Flujo de diseño de un sistema embebido. Extraida de [4]

2.1.1. Flujo de diseño hardware

Para seguir de la forma más correcta posible el flujo de diseño hardware se ha utilizado como referencia para toda la información del flujo de diseño hardware, el manual de Xilinx de [5]. Las herramientas Vivado proporcionan flujos específicos para la programación, basados en el procesador. El IDE de Vivado utiliza el integrador de IP con pantallas de conectividad gráfica para especificar el dispositivo, seleccionar periféricos, y configurar los ajustes de hardware.

Se puede utilizar Vivado IP integrator para capturar la información de la plataforma de hardware en formato XML junto con otros archivos de datos para desarrollar diseños para procesadores Xilinx. El diseño de software herramientas de diseño de software utilizan el XML para distintas funcionalidades.

La figura 2.3 ilustra el flujo de herramientas para el hardware integrado de un dispositivo Zynq UltraScale+ (MPSoC).

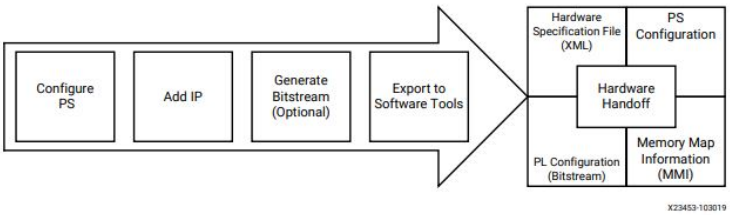


Figura 2.3: Flujo de diseño hardware. Extraida de [5]

2.1.2. Flujo de diseño software

En cambio, para el flujo de diseño software nos guiaremos por el manual de Xilinx [6]. La plataforma de software unificada Vitis es una nueva herramienta que combina todos los aspectos del desarrollo de software de Xilinx en un entorno unificado. La plataforma de software Vitis admite tanto el flujo de desarrollo de software embebido Vitis, para los usuarios del kit de desarrollo de software (SDK) de Xilinx que buscan pasar a la tecnología de próxima generación, como el flujo de desarrollo de aceleración de aplicaciones Vitis, para los desarrolladores de software que buscan utilizar lo último en aceleración de software basado en FPGA de Xilinx. Este documento analiza el flujo de desarrollo del software embebido y el uso del kit de desarrollo del núcleo de Vitis. El entorno de desarrollo integrado (IDE) Vitis forma parte de la plataforma de software unificada Vitis. El IDE de Vitis está diseñado para el desarrollo de aplicaciones de software embebido dirigidas a los procesadores embebidos de Xilinx. El IDE Vitis funciona con diseños de hardware creados con Vivado Design Suite. El IDE Vitis se basa en el estándar de código abierto Eclipse.

2.2. El proyecto PYNQ

Se ha analizado la documentación de la página oficial, a la cual podemos acceder en [7], en ella se verá que el objetivo principal de PYNQ , es facilitar a los diseñadores de sistemas embebidos la explotación de las ventajas únicas de los dispositivos Xilinx en sus aplicaciones. En concreto, PYNQ permite a los arquitectos, ingenieros y programadores que diseñan sistemas embebidos utilizar los dispositivos Zynq UltraScale+ (MPSoC), sin tener que utilizar herramientas de diseño tipo ASIC para diseñar circuitos lógicos programables. PYNQ logra este objetivo de tres maneras.

2.2.1. Overlays de PYNQ

Los circuitos lógicos programables se presentan como bibliotecas de hardware denominadas overlays o capas, la documentación relativa a ellas está en [8]. Estas overlays son análogas a las bibliotecas de software. Un ingeniero de software puede seleccionar la overlay que mejor se adapte a su aplicación. Se puede acceder a la overlay a través de una Application Programming Interface (API) . La creación de una nueva overlay sigue requiriendo ingenieros con experiencia en el diseño de circuitos lógicos programables. Sin embargo, la diferencia clave es el paradigma de construir una vez y reutilizar muchas veces. Las overlays, al igual que las bibliotecas de software, están diseñadas para ser configuradas y reutilizadas tantas veces como sea posible en muchas aplicaciones diferentes.

El dispositivo programable de Xilinx Zynq UltraScale+ (MPSoC) es un SoC basado en un procesador ARM Cortex-A9 de doble núcleo (denominado Processing System (PS)), integrado con la estructura de la FPGA (denominada Programmable Logic (PL)). El subsistema PS incluye una serie

de periféricos dedicados (controladores de memoria, USB, Uart, IIC, SPI, etc.) y puede ampliarse con IP de hardware adicional en una superposición PL .

Los overlays, o bibliotecas de hardware, son diseños de FPGA programables/configurables que extienden la aplicación del usuario desde el sistema de procesamiento del Zynq UltraScale+ (MPSoC) a la lógica programable. Los overlays pueden utilizarse para acelerar una aplicación de software o para personalizar la plataforma de hardware para una aplicación concreta. El diagrama de bloques de los overlays utilizados por PYNQ es el que podemos ver en la figura 2.4.

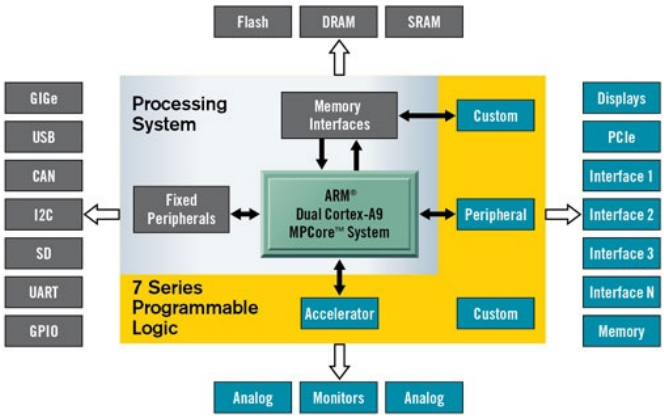


Figura 2.4: Diagrama de bloques de los overlays de PYNQ. Extraida de [8]

Por ejemplo, el procesamiento de imágenes es una aplicación típica en la que las FPGA pueden proporcionar aceleración. Un programador de software puede utilizar un overlays de forma similar a una biblioteca de software para ejecutar algunas de las funciones de procesamiento de imágenes (por ejemplo, detección de bordes, umbralización, etc.) en la estructura de la FPGA . Los overlays pueden cargarse en la FPGA de forma dinámica, según sea necesario, al igual que una biblioteca de software. En este ejemplo, las funciones de procesamiento de imágenes separadas podrían implementarse en diferentes overlays y cargarse desde Python bajo demanda.

2.2.2. Entorno de Python

Siendo analizada la información obtenida de [9], se verá que PYNQ utiliza Python para programar tanto los procesadores integrados como las overlays. Python es un lenguaje de un gran "nivel de productividad". Hasta la fecha, C o C++ son los lenguajes de programación integrados más comunes. En cambio, Python eleva el nivel de abstracción de la programación y la productividad del programador. Sin embargo, estas opciones no se excluyen mutuamente. PYNQ utiliza CPython, que está escrito en C, e integra miles de bibliotecas C y puede ampliarse con código optimizado escrito en C. Siempre que sea práctico, debe utilizarse el entorno Python más productivo, y siempre que la eficiencia lo exija, puede utilizarse código C de nivel inferior.

2.2.3. Notebooks de Jupyter

PYNQ es un proyecto de código abierto que pretende funcionar en cualquier plataforma informática y sistema operativo. Este objetivo se consigue adoptando una arquitectura basada en la web, que además es agnóstica en cuanto a navegadores. Incorporamos la infraestructura del Jupyter notebook de código abierto para ejecutar un núcleo de Interactive Python (IPython) y un servidor web directamente en el procesador ARM del dispositivo Zynq UltraScale+ (MPSoC), esta información la hemos obtenido de la documentación vista en [10]. El servidor web accede al kernel o núcleo a través de un conjunto de herramientas basadas en el navegador que proporcionan un panel de control, un terminal bash, editores de código y cuadernos Jupyter. Las herramientas del navegador se implementan con una combinación de JavaScript, HTML y CSS y se ejecutan en cualquier navegador moderno.

Jupyter Notebook es un entorno informático interactivo que permite a los usuarios crear documentos de cuaderno, esto nos permite tener y utilizar código en directo, widgets interactivos, gráficas, texto narrativo con el que aclarar y comentar dentro del propio código, ecuaciones matemáticas, imágenes y por último lo que más vamos a utilizar, vídeo. Un notebook de Jupyter está compuesto por la combinación de tres componentes.

La aplicación web del cuaderno, una aplicación web interactiva para escribir y ejecutar código de forma interactiva y crear documentos de cuaderno. Núcleos, procesos independientes iniciados por la aplicación web del cuaderno que ejecutan el código de los usuarios en un lenguaje determinado y devuelven la salida a la aplicación web del cuaderno. El núcleo también se encarga de cosas como los cálculos para los widgets interactivos, la finalización de pestañas y la introspección. Y por último, documentos de cuaderno, estos son documentos autocontenidos que contienen una representación de todo el contenido de la aplicación web de cuaderno, incluidas las entradas y salidas de los cálculos, el texto narrativo, las ecuaciones, las imágenes y las representaciones multimedia de los objetos. Cada documento de cuaderno tiene su propio núcleo.

Tras haber mencionado estas tres maneras cabe destacar que PYNQ es el primer proyecto que combina ciertos elementos para simplificar y mejorar el diseño de Array Processor (AP) on a chip (SoC) (APSoC) . Estos elementos son, un lenguaje de productividad de alto nivel (Python en este caso), overlays de FPGA con amplias API expuestas como bibliotecas de Python, una arquitectura basada en la web servida desde los procesadores integrados y el marco de trabajo de Jupyter Notebook desplegado en un contexto embebido. Podemos ver un resumen de la estructura del proyecto PYNQ en la figura 2.5.

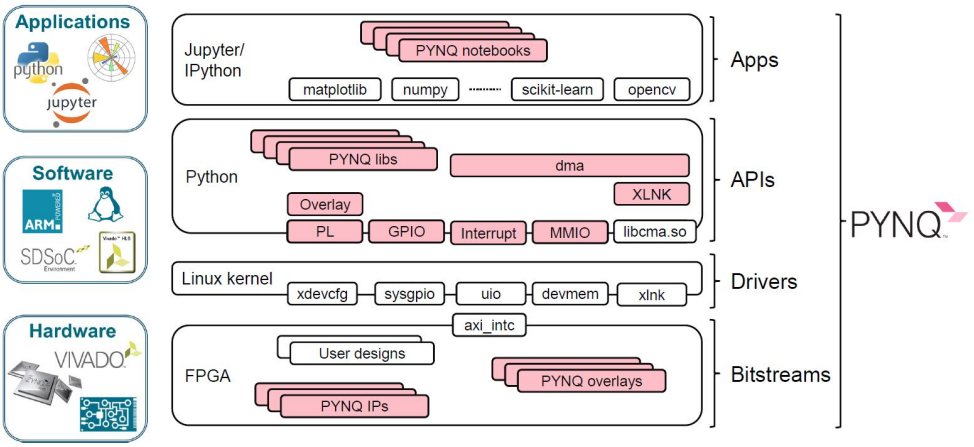


Figura 2.5: Esquema del proyecto PYNQ. Extraida de [11]

2.3. La plataforma ZCU104

Entre las muchas placas de desarrollo que han sido diseñadas por Xilinx, PYNQ soporta las basadas en Zynq y MPSoC, tras haber utilizado las más básicas de entre ellas para familiarizarnos con el proyecto y con su funcionamiento, como la PYNQ-Z1 y la PYNQ-Z2, se ha decidido, tanto por sus características y funcionalidades así como su capacidad de procesamiento superior a las mencionadas anteriormente en este mismo párrafo, el utilizar la placa ZCU104 para este proyecto.

La ZCU104 proporciona una plataforma de prototipado rápido para el mercado de visión embebida de bajo coste utilizando el dispositivo XCZU7EV-2FFVC1156. El ZU7EV contiene periféricos de bloque duro de PS mostrados a través de la interfaz de Entrada/Salida multiusos y varias FPGA programables, de alta densidad y de alto rendimiento.

El kit de evaluación ZCU104 de Xilinx permite a los diseñadores impulsar diseños para aplicaciones de visión integradas como vigilancia, Advanced Driver Assistance Systems (ADAS) , visión artificial, Augmented Reality (AR) , drones e imágenes médicas. Este kit, al cual podemos acceder en el enlace bibliográfico [12], incluye un dispositivo Zynq UltraScale + MPSoC EV con códec de video y es compatible con muchos periféricos e interfaces comunes para casos de uso de visión integrada. El dispositivo ZU7EV incluido está equipado con un procesador de aplicaciones Arm™ Cortex™-A53 de cuatro núcleos, un procesador en tiempo real Cortex-R5 de doble núcleo, una unidad de procesamiento de gráficos Mali-400 MP2, video H.264/H.265 compatible con 4KP60 códec y FinFET de 16 nm + lógica programable.

Ahora se van a ver cuáles son las principales características que trae incorporadas la placa, y que también tenemos listadas en la documentación [12]. El análisis de esta placa no es el objetivo del trabajo por lo que estas características serán mencionadas de forma puntual y no vistas en profundidad.

Principales características de la ZCU104

- Elementos de configuración
  - USB-JTAG FT4232H; Dual Quad-SPI flash memory; MicroSD Card
- Memoria
  - PS DDR4 64-bit Component; Quad-SPI flash; Micro SD card slot
- Elementos de control y de Entrada/Salida
  - 4x directional pushbuttons; DIP switches; PMBUS, clocks, and I2C bus switching; USB2/3
- Conectores de expansión
  - FMC LPC (1x GTH); 3 PMOD connectors; PL DDR4 SODIMM Connector – 64 bit
- Comunicación y redes
  - USB-UARTs with FT4232H JTAG/3xUART Bridge; RJ-45 Ethernet connector; SATA (M.2) for SSD access
- Display
  - HDMI 2.0 video input and output (3x GTH); DisplayPort (2x GTR)
- Reloj
  - Programmable clocks, System clock, user clock; Jitter attenuator
- Alimentación
  - 12V wall adaptor or ATX

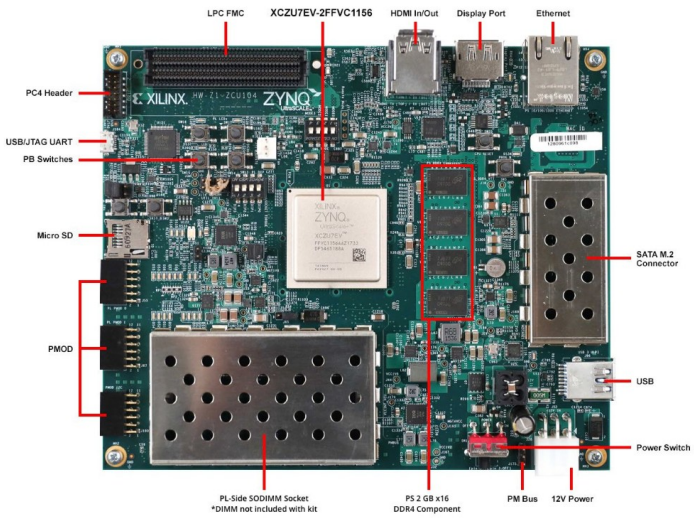


Figura 2.6: Imagen de una ZCU104 y sus componentes. Extraido de [12]

2.4. Uso de HLS para diseño hardware

La programación de una FPGA , tal y como vamos a realizar con nuestra placa ZCU104, se realiza mediante la carga de un fichero binario denominado bitstream en la propia FPGA , este fichero con-

tendrá la información de configuración generada durante la fase de diseño del hardware, y se bajará a la placa implementando dicha información.

2.4.1. Diseño hardware

Esta información se introducirá en el fichero mediante herramientas de diseño hardware, el desarrollo de estas herramientas lo podemos ver en más profundidad en [13], donde tenemos acceso a todo lo relativo al uso de High-Level Synthesis (HLS) para diseño hardware en más profundidad. Siguiendo el flujo tradicional de diseño, el cual consiste en una fase de definición de la lógica a implementar seguida de una fase de traducción a elementos lógicos para terminar con el posicionamiento y conexión de estos mismos sobre la FPGA .

2.4.2. Hardware Description Language

El Hardware Description Language (HDL) es el lenguaje que se emplea para definir los circuitos necesarios, todo ello utilizando una abstracción al Register Tranfer Language (RTL) , esto quiere decir que trabaja mediante el seguimiento del flujo de las señales. Distintas maneras de diseñar un circuito son la definición de su comportamiento respecto a su entrada y salida, de una manera estructural y jerarquizada o con una solución que combine las dos anteriores, los lenguajes más utilizados para esto son VHDL y Verilog.

A pesar de que HDL se encuentre dos niveles por encima del mínimo nivel de abstracción posible, este se considera un lenguaje de bajo nivel. Aunque es posible diseñar sobre FPGA a nivel lógico mediante esquemáticos cuya base son las puertas lógicas, en cuanto nos encontramos con circuitos de cierta complejidad ya no es viable utilizar este sistema. Una de sus principales desventajas es la pérdida de optimización al aumentar de nivel de abstracción, pero esta queda compensada con creces gracias al gran aumento de productividad que nos da este mismo aumento de abstracción.

La baja eficiencia de la realización de diseños HDL hace que cada vez más, en el mercado actual, se busque el uso de niveles de abstracción, y por lo tanto de lenguajes, de nivel alto. Esto se debe a que los tiempos de desarrollo son muy altos, mientras que hay una gran escasez cantidad de gente especializada con conocimiento de lenguajes de bajo nivel de abstracción en este campo. Una de las posibles soluciones y que muchas empresas barajan es utilizar lenguajes de alto nivel tales como C, contrarrestando así todas las desventajas que hemos visto previamente, esto nos lleva al siguiente punto de este trabajo.

2.4.3. High-Level Synthesis

La forma de poder trabajar con lenguajes de alto nivel tales como C, y que de estos podamos generar los circuitos que se implementen sobre la FPGA , utilizaremos herramientas de HLS . La funcionalidad de estas herramientas es traducir los códigos realizados en lenguajes de alto nivel a HDL , el proceso en el que hace esta traducción es una conexión muy parecida a la que haría un compilador. El código HDL generado automáticamente tendrá exactamente la misma funcionalidad que la que se le ha dado al código en lenguaje de alto nivel, es una traducción exacta, la única diferencia es que el código HDL operará sobre un circuito hardware específico, esto conlleva la desventaja antes comentada que no estará tan bien optimizado como si se hubiese hecho en HDL desde el comienzo, es de esperar que haya un aumento en la latencia o en la cantidad de recursos utilizados, o de ambos al mismo tiempo.

Durante el transcurso de este proyecto las herramientas que hemos utilizado han sido Vivado y Vivado HLS, ambas desarrolladas por Xilinx al igual que la ZCU104, la placa que utilizaremos, un esquema muy básico del flujo de diseño que se sigue al realizar HLS es el que podemos ver en la figura 2.7.

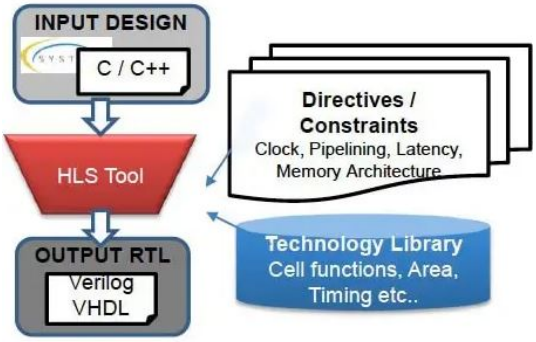


Figura 2.7: Flujo de diseño HLS. Extraido de [14]

El procedimiento de HLS en Vivado HLS no se hace de forma directa, esto quiere decir que podemos alterar y añadir directivas antes de pasar a la síntesis modificando así el hardware obtenido. Este IDE tiene como base comandos en lenguaje Tool Command Language (TCL) para la ejecución de las funciones que posee la aplicación, por lo que las directivas antes mencionadas de optimización en HLS se definirán en este mismo lenguaje incluyéndose así mismo en un fichero de extensión .tcl, o como veremos más adelante en forma de pragmas en nuestro código fuente.

3

DISEÑO DE REFERENCIA

Tal y como se ha definido previamente, PYNQ es un proyecto de código abierto que busca hacer minimizar la complejidad del diseño de sistemas embebidos, nuestro objetivo es modificar el diseño base, el diseño de referencia para demostrar este concepto.

Para ello antes de comenzar con dichas modificaciones se va a intentar comprender lo máximo posible este diseño base, sobre todo en su funcionalidad de procesamiento de video, ya que es lo que vamos a modificar posteriormente en la sección 4.3.

Lo primero que se hará será descargar el diseño de referencia, el cual nos es dado en la propia página de PYNQ por sus diseñadores, y que podemos ver en la referencia bibliográfica [15], hay una enlace directo al github donde está subido y al cual podemos acceder pinchando aqui. Todo lo relativo a este capítulo se ha obtenido analizando y probando minuciosamente este diseño de referencia dado por Xilinx.

Una vez ya descargado, y accediendo su directorio principal mediante el sistema de archivos, buscaremos el archivo makefile de la placa que vamos a utilizar, en este caso la ZCU104, el cual se encuentra en "PYNQ-master/boards/ZCU104/base". Abrir en este directorio un terminal y realizar el comando "make all", nos generará el proyecto en una nueva carpeta dentro de base, la cual se llamará por defecto igual, base, y nos realizará la compilación al completo del mismo.

Este proceso toma algo de tiempo, ya que tiene que generar y compilar el proyecto completo, pero una vez creado el proyecto, podremos abrirlo utilizando Vivado. Una vez abierto con Vivado abriremos el diseño de bloques, lo cual nos dará una vista global de todos los elementos que conforman el proyecto así como de las conexiones que hay entre ellos, tal y como se muestra en la figura 3.2.

3.1. Diseño de bloques

En esta sección vamos a analizar un poco más en profundidad el flujo de diseño junto al diseño de bloques del proyecto de referencia, así como las interconexiones entre sus elementos, iremos centrándonos sobre todo en el bloque de procesamiento de video, ya que es dentro del cual desarrollaremos

nuestro proyecto.

Como podemos apreciar en la figura 3.2, tenemos el diseño de bloques del proyecto al completo, un resumen más comprensible y analizable de todo este diseño es el de la figura 3.1. En esa figura se ve de forma muy simplificada todo el diseño antes visto.

Analizar la figura 3.1 nos va a aportar más entendimiento de manera más rápida acerca del diseño por lo que es lo que haremos, aclarar que esta figura es la misma que la 3.2 pero resumida y vista de forma más teórica. Lo primero que vemos es que el systema de procesamiento Zynq se comunica con cada uno de los bloques que conforman la lógica programable, dentro de estos bloques está el que subsistema de video, que es el que nos va a interesar para nuestro proyecto.

Los procesadores Pmod IO son MicroBlazes de PYNQ, esto quiere decir que son subsistemas de procesadores blandos MicroBlaze dedicados, que permiten conectar al sistema varios periféricos con diferentes estándares de Entrada o Salida bajo demanda. Esto permite al programador de software utilizar una amplia gama de periféricos con diferentes interfaces y protocolos. Al utilizar un PYNQ MicroBlaze, se puede utilizar la misma overlay para soportar diferentes periféricos sin necesidad de una overlay diferente para cada periférico.

La ZCU tiene 2 Pmod PYNQ MicroBlaze, uno conectado al PMOD 0 y el otro al PMOD 1, conectando a cada tipo de interfaz correspondiente. Hay que tener en cuenta que el PMOD 2 está conectado al PS I2C y no está disponible como puerto Pmod de propósito general.

La placa tiene 4 LEDs de usuario, 4 dip-switches y 4 pulsadores. Estos dispositivos de entrada y salida son controlados a través de controladores AXI GPIO en la lógica programable. En la overlay base de la ZCU104, estos dispositivos de entrada y salida están enrutadas al PS GPIO, y pueden ser controladas directamente desde Python.

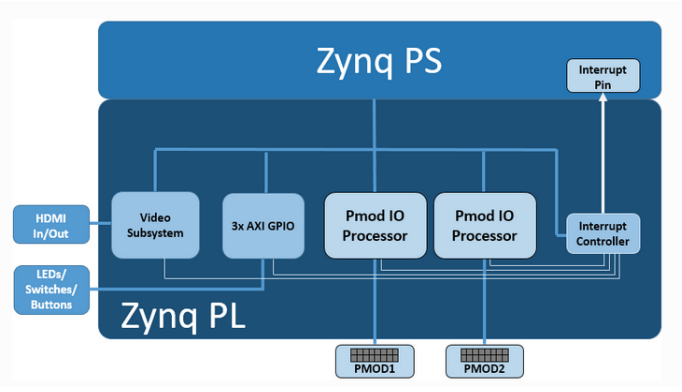


Figura 3.1: Figura del diseño de bloques simplificado para la ZCU104. Extraida de [16]

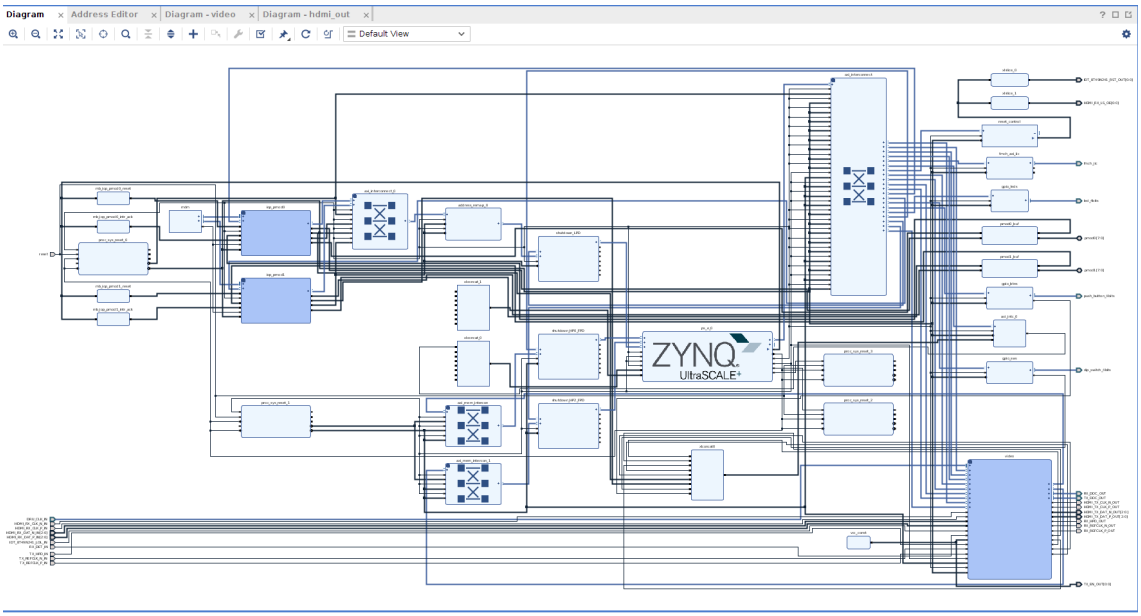


Figura 3.2: Figura del diseño de bloques completo del proyecto PYNQ para la placa ZCU104. Figura propia

3.2. Diseño del bloque de video

Una vez se ha visto el diseño global más a fondo en la sección 3.1, se va a centrar en el bloque de video, el bloque de video del diseño se puede ver en la figura 3.3. Este mismo tiene una conexión directa con el sistema de procesamiento y está formado princpalmente por dos grandes bloques, el HDMI de entrada y el HDMI de salida. El sistema en sí posee varios relojes o CLKs, de los cuales solo uno va a controlar este bloque, dándole esa velocidad de procesamiento necesaria para cualquier trabajo relacionado con procesamiento de video

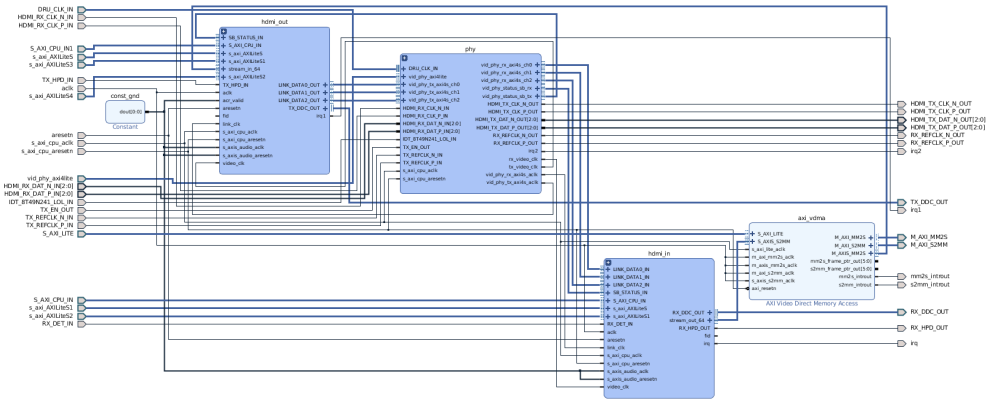


Figura 3.3: Figura del diseño de bloques del video del proyecto. Figura propia

3.3. Diseño de bloques del HDMI de salida

Dentro del bloque del HDMI de salida es donde se va a realizar la modificación en que consiste el proyecto, por ello aquí se va a implementar el IP core con la nueva función, antes de pasar a esa implementación se va a analizar todo el diseño de bloques interior a este HDMI de salida.

Como se ve en la figura 3.4, el diagrama de bloques recibe los píxeles de dos en dos, y estos sufren varias transformaciones antes de pasar a un último bloque de frontend que ya los remite a la salida.

Estos bloques son el desempaquetamiento de pixel, el cual actúa de forma inversa a un bloque similar, pero de empaquetamiento que hay en el HDMI de entrada, el de conversión de color, que actua de forma muy similar a la función que vamos a desarrollar, un bloque con nuestra función que como veremos más adelante hemos implementado desde cero (este bloque no está en el diseño original), un pequeño bloque de registro y por último antes del frontend un bloque de reordenamiento de píxeles.

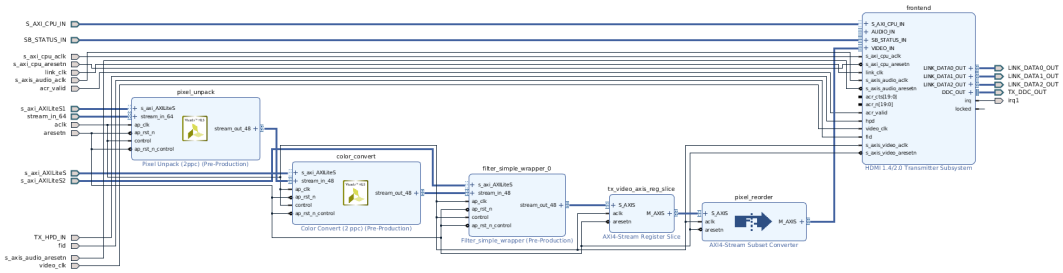


Figura 3.4: Figura del diseño de bloques del HDMI de salida. Figura propia

4

IMPLEMENTACIÓN DEL PROYECTO

Una vez se ha visto el marco teórico que forma la base de nuestro proyecto en el capítulo 2, y se ha analizado con algo más de detalle el diseño de referencia en el capítulo 3, en este capítulo se va a desarrollar paso a paso la implementación del proyecto en que consiste este trabajo.

Este es el principal objetivo de este proyecto y consiste en la modificación del overlay hardware del diseño de referencia añadiendo un nuevo IP core. Este nuevo bloque tendrá una o varias funcionalidades elegidas por nosotros que permitan apreciar la multitud de posibilidades que ofrece este proyecto.

Se han seleccionado como funcionalidades a implementar, por el nuevo IP core, la alteración de un video saliente por el puerto HDMI de salida de nuestra placa ZCU104. Las distintas funcionalidades consisten en un filtrado de imagen que permite modificar los colores en función de un parámetro controlable mediante un lenguaje de alto nivel, en este caso python.

Así, tal y como se ha visto en los objetivos, se apreciará como haciendo un cambio muy simple en el hardware, tan solo se está añadiendo un nuevo IP core o bloque al diseño de referencia con una funcionalidad cualquiera pudiendo ser elegida, se ha logrado controlar este IP core desde alto nivel. Sin ningún problema, se podrá elegir entre sus funcionalidades tan solo modificando un parámetro desde un notebook de Jupyter mediante Python.

En el apartado 4.2 se seguirá el desarrollo del nuevo IP core, el código que ejecutará el nuevo bloque hardware, una vez se ha explicado este código, se va a pasar a su implementación dentro del diseño de referencia en el apartado 4.3, alterandose este mismo para que forme parte de él.

4.1. Configuración de la ZCU104

Para realizar la configuración inicial de la ZCU104 se han seguido las instrucciones presentes en la página de los diseñadores de PYNQ , a la cual se puede acceder pulsando aqui, dentro de este tutorial se ha seguido, concretamente, un video que hace una guía completa y que está publicado en el mismo tutorial, a este video se puede acceder directamente pulsando aqui.

EL tutorial consiste en descargar la imagen de PYNQ correspondiente a la ZCU104 e introducirla en la tarjeta SD de memoria que va en la propia placa, tras esto vuelve a incroporar la tarjeta SD a la placa y realiza todas las conexiones de los cables coloca en la posición inicial todos los switches de nuestra ZCU104.

El último paso es el proceso de encendido y conexión, aquí se ha encendido la placa, comprobado que se ha iniciado correctamente y que el bitstream se ha cargado sin ningún problema. Seguidamente se ha accedido al entorno web de jupyter y se han realizado algunas pruebas de control con un notebook nuevo para asegurar que todo se haya realizado de forma correcta.

4.2. Diseño inicial del IP core

Esta sección consiste en la implementación del código que va a formar el nuevo IP core, este código contendrá todas las posibles funcionalidades que se desean añadir al proyecto y posteriormente poder controlar desde alto nivel.

Este proceso se realizará enteramente en Vivado HLS, herramienta que ya se ha visto previamente en la sección 2.4, mediante Vivado se implementará un filtro convolucional en C, que formará nuestro IP core.

Este filtro consiste en una función denominada **filter_simple_wrapper**, que recibe como entrada un streaming de 48 bits, una vez tiene este streaming lo pasa por dos funciones distintas, estas son **add_gray** y **filter_simple**. Hemos diseñado un filtro que trabajo con los píxeles recibidos de dos en dos, de ahí que el streaming sea de 48 bits, y tal y como veremos más adelante vamos a operar sobre ambos al mismo tiempo en cada una de las funciones que lo componen.

Código 4.1: En esta figura se presenta el código correspondiente a la funcion filter_simple_wrapper que contiene las dos funciones mencionadas previamente.

```
void filter_simple_wrapper(stream<video_stream>& stream_in_48, stream<video_stream>&
      stream_out_48, ap_uint<32>& my_param){
#pragma HLS CLOCK domain=default
#pragma HLS DATAFLOW
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE s_axilite register port=my_param clock=control
#pragma HLS INTERFACE axis port=stream_in_48 register
#pragma HLS INTERFACE axis port=stream_out_48 register

static stream<internal_stream> stream_internal64("stream_internal64");
    add_gray(stream_in_48, stream_internal64);
    filter_simple(stream_internal64, stream_out_48, my_param);
}
```

Dentro de nuestra función principal se va a desglosar e ir explicando una a una las dos funciones

que lo componen, como se puede ver en el código apartado 4.2, la primera función a la que se llama desde la función principal es **add_gray**. Esta función tiene como objetivo pasar los bits que recibe en forma RGB a escala de grises, primero se comprueba los bits de salto de línea y fin de imagen, luego pasa a la escala de grises, y por último comprueba que no haya overflow.

La segunda función que conforma la función principal se denomina **filter_simple**, su funcionalidad es leer un parámetro que se ha creado, y en función de ese parámetro alterar los píxeles que asociaremos a sus respectivos colores. Esto quiere decir que si, por ejemplo, se decide mostrar tan solo los píxeles en rojo se anularán todos los píxeles recibidos menos los correspondientes al color rojo. Esto permite modificar cualquier color del streaming de salida así como al estar trabajando de a dos bits, seleccionar si se desea partir la imagen alterando los coeficientes de color de tan solo la mitad de los bits que se reciben.

Para información más detallada acerca del código de las funciones que conforman la función principal, remitirse al apéndice A.

4.3. Modificación del overlay hardware

En esta sección se verá como, una vez se ha creado el IP core, se va a añadir al diseño de referencia como un bloque más dentro de este mismo. Para eso se van a seguir una serie de pasos recogidos en esta sección.

Para ello se va a documentar cada paso mediante capturas, rutas y explicaciones. Este proceso está basado en uno más simple publicado como tutorial en la documentación de PYNQ en su página web, para acceder a él seleccionar aquí. Todo este proceso se ha realizado utilizando las herramientas de Xilinx Vivado y Vivado HLS.

Paso 1. Preparación

- Se comenzará descargando el proyecto master de PYNQ, directamente del github de Xilinx pinchando aquí
- Para continuar se extraerá y situará esta carpeta en el mismo directorio que el proyecto hls, con el IP core que se desea implementar.
- Dentro del proyecto master de PYNQ interesa la placa ZCU104 por lo que se accederá al directorio base de la placa, donde se tendrá un fichero makefile, el cual al ser ejecutado en un terminal, generará una nueva carpeta base con el proyecto al que se desea añadir el IP core.

Paso 2. Exportación del IP core

- Dentro de vivado hls, en el proyecto, se pulsará sobre la opción “Export RTL”.
- Una vez finalizada la exportación se accederá a las soluciones, mediante la búsqueda del fichero hardware del IP core, el cual se llamará xfilter_simple_wrapper_hw.h. Ahí se encontrarán los registros con las direcciones que se ven en la figura 4.1.

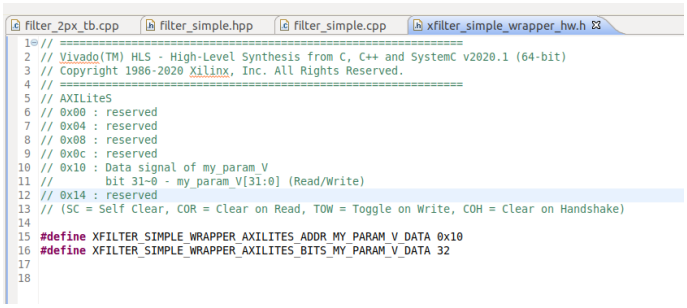


Figura 4.1: Figura de los registros de offset resultantes. Figura propia

Paso 3. Añadir IP core al proyecto master

- Una vez exportado se abrirá, mediante Vivado, el proyecto master de la ZCU104 que se ha generado en el Paso 1.
- Ya abierto en la parte izquierda se pulsará sobre “open block design”, obteniendo la vista del diseño ya visto en el capítulo 3, se podrá volver al diagrama de bloques del diseño en la figura 3.1.
- Dentro del diseño global, se va a trabajar con el bloque de video, por lo que se accederá pulsando dos veces sobre él obteniendo lo que se apreció en la sección 3.2, y en la figura ya vista 3.3.
- Dentro de este se ha decidido añadir el IP core solo en el HDMI de salida, el hdmi_out por lo que se abrirá también pulsando sobre el dos veces.
- Una vez se ha completado este proceso y se ha ubicado donde se quiere añadir el nuevo bloque, se añadirá el IP al proyecto, para ello se pulsará sobre “settings”, en PROYECT MANAGER, y dentro de los proyect settings se abrirá IP/Repository, aquí se seleccionará el icono del + y se buscará la ruta de nuestro proyecto de Vivado HLS, el cual se ha denominado prj_hls.
- Tal y como se aprecia en la imagen 4.2, Vivado ha encontrado el IP, donde se ve que su top function se llama ‘Filter_simple_wrapper’.

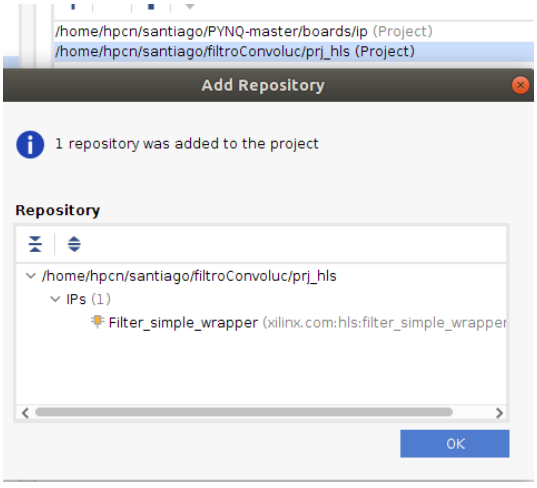


Figura 4.2: Figura con el cuadro de opciones de vivado que nos permite añadir una IP nueva a nuestro diseño. Figura propia

- Ahora ya se puede utilizar el bloque por lo que se volverá al diseño de bloques del hdmi_out, y se pulsará sobre el símbolo + para añadir el bloque, se ha buscado el filter_simple_wrapper_0 y se ha mostrado sobre el resto del diseño de bloques, aquí se pueden realizar las conexiones de forma manual o lanzarse la opción de “Run

automation Block Desing".

- Al utilizarse dos relojes distintos se podría llegar a mantener señales a distintas frecuencias, y por lo tanto realizar las conexiones de forma manual podría tener su atractivo, pero en este caso es deseable utilizar el mismo reloj para todas las señales por lo que se utilizará la opción del diseño automático.
- Nos queda un diseño de bloques final del hdmi_out como la figura 4.3

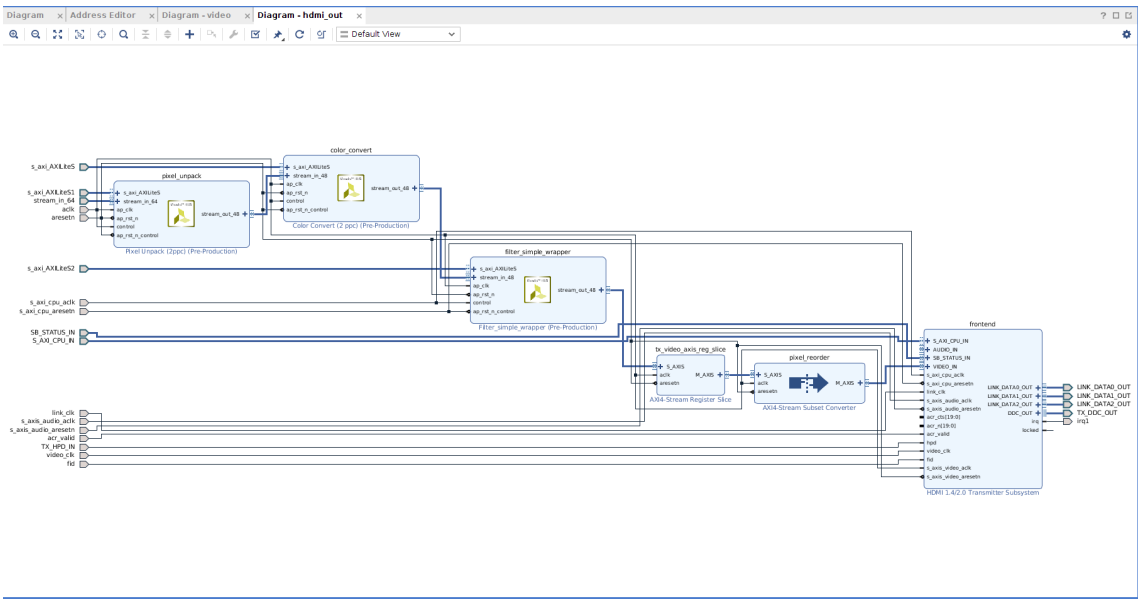


Figura 4.3: Figura del diseño de bloques del hdmi de salida

- El siguiente paso ha sido generar el wrapper para posteriormente poderse generar el fichero bitstream, el cual se pasará a la placa, para ello se ha ido a la pestaña Source del block design y se ha seleccionado, de base_I(base.bd), "Generate HDL wrapper".
- Una vez hecho esto se ha seleccionado la opción "Generate Bitstream" y dejado que Vivado genere de forma automática el fichero bitstream.

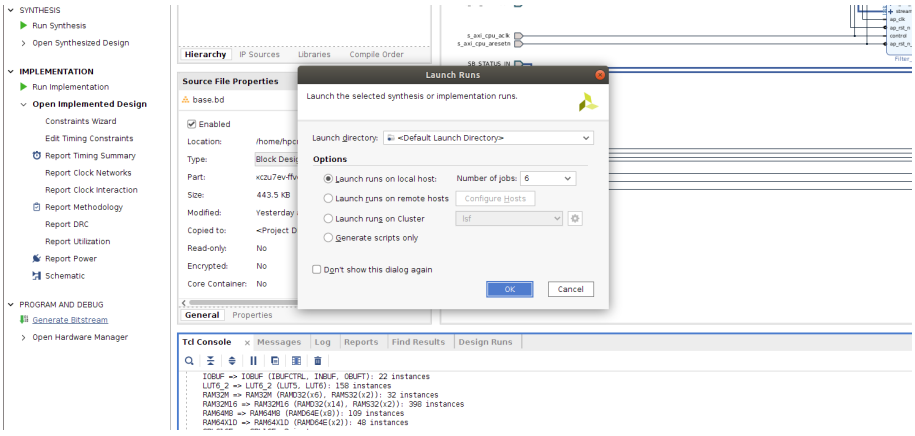


Figura 4.4: Figura de generación del bitstream para su posterior descarga en la placa. Figura propia

Paso 4. Exportar a la ZCU104

- Tras finalizar la generación del bitstream se exportará a la placa, para ello se ha comenzado exportando el diseño de bloques, esto se ha realizado en “File/Export/Export Block Design. . . ”.

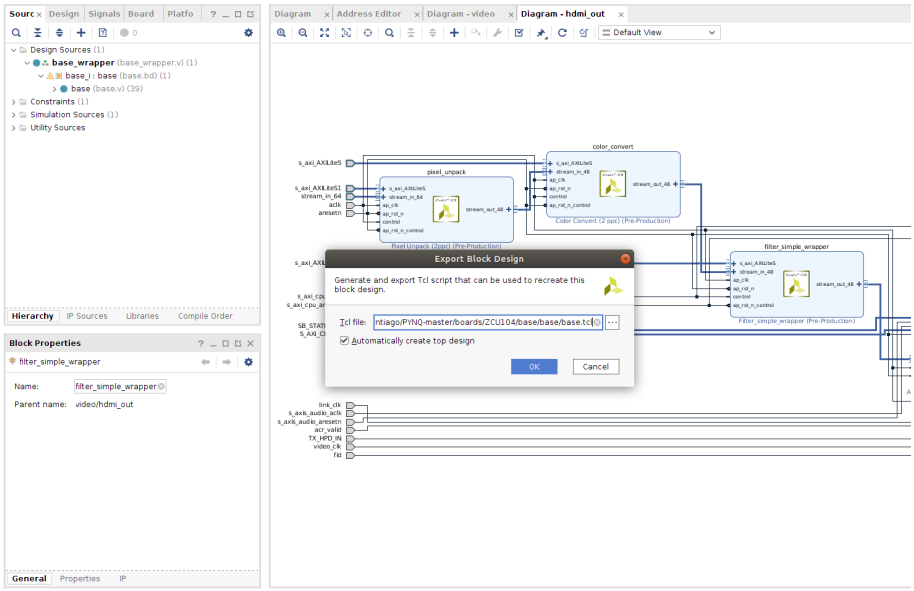


Figura 4.5: Figura del cuadro de opciones de vivado para la exportación del Block Design. Figura propia

- Accediendo a los ficheros del proyecto, junto al archivo .xpr en la ruta del proyecto se va a encontrar el diseño de bloques exportado como un fichero llamado base.tcl, accediendo a la carpeta “base.runs/impl_1”, también situada en el directorio del proyecto, se encontrará el fichero bitstream que se generó en el paso anterior, y que se denomina base-wrapper.bit, a su vez debido a la última actualización de pynq es requisito que también se pase el fichero .hwh por lo que también lo se pasará a los ficheros de la placa.
- Una vez se han localizado los ficheros, serán renombrados el fichero de bitstream y el .hwh para que tenga el mismo nombre que nuestro .tcl, se denominarán base por abreviar.
- Lo siguiente a realizar será copiar estos tres archivos a la tarjeta SD de la placa ZCU104.
- Se encenderá la placa y se dejará que cargue los ficheros de forma correcta, esto se confirma al comprobar que los LEDs están en verde y encendidos, lo cual nos indicará que está correctamente iniciada y lista para su funcionamiento, tal y como se muestra en la figura 4.6.
- Una vez completado el arranque, se va a acceder a su sistema de archivos para pasar los ficheros que componen el proyecto.
- Accederemos a los ficheros de la placa desde el sistema de archivos mediante sftp, este proceso se realiza yendo a “Other Locations” y escribiendo sftp://192.168.5.171/ (dirección de la placa en nuestra red).
- Ahora se ha seguido la siguiente ruta “192.168.5.171/pynq/overlays”, en ella se ha creado una nueva carpeta llamada filter y se han copiado el bitstream, el .hwh y el .tcl antes creados a esta carpeta. PYNQ ya tiene acceso a este overlay.

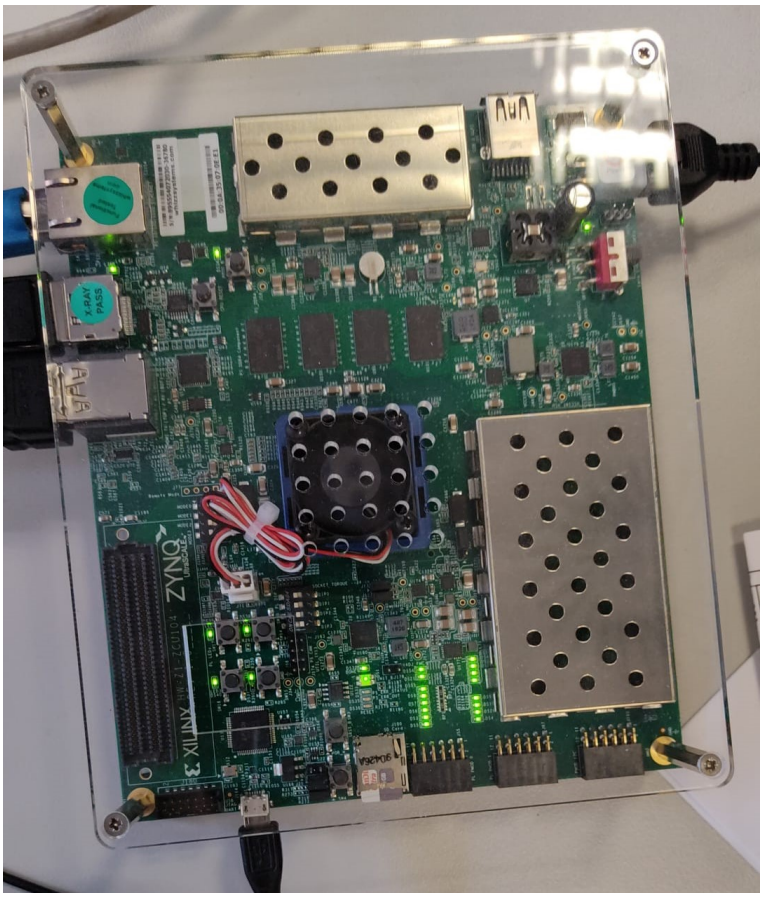


Figura 4.6: Figura de nuestra placa ZCU104 encendida y con los leds indicativos de correcto funcionamiento activados. Figura propia

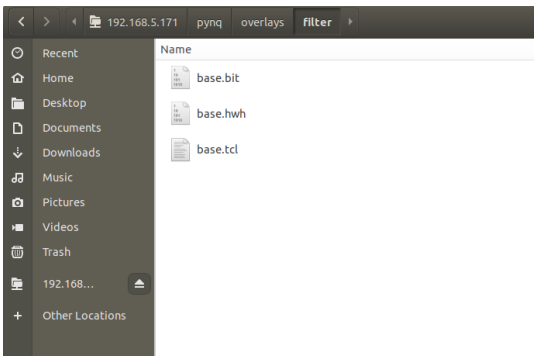


Figura 4.7: Figura de la carpeta filter creada en el sistema de archivos de la placa con los tres ficheros ya descargados. Figura propia

Paso 5. Uso del Overlay desde Jupyter

- Este paso ha consistido en probar el Overlay que se ha añadido, esto se hará accediendo a la dirección de la placa en el navegador web, con el puerto 9090, se solicitará una contraseña que será la misma que se ha utilizado para acceder por ssh, xilinx.

Figura 4.8: Figura de la página de inicio al acceder al entorno web de jupyter. Figura propia

- Se accederá a “base/video” y ahí se creará un nuevo Notebook de Jupyter desde donde se cargará nuestro Overlay y se probará su correcto funcionamiento, para crearlo se pulsará sobre el botón ‘New’ y ‘Python 3’, y se llamará ‘hdmi-Santiago-FinalOverlay’ tal y como se ve en la figura 4.9.

Figura 4.9: Figura de creación de un nuevo notebook en jupyter donde realizaremos todas las pruebas para verificar funcionamiento de nuestro IP core. Figura propia

- Ya se ha creado y se va a cargar el Overlay mediante el código que se ve en la figura 4.10.

Figura 4.10: Figura de la carga del nuevo Overlay dentro de una variable en el notebook de jupyter. Figura propia

5 EVALUACIÓN Y RESULTADOS

En este capítulo se van a ver las pruebas y los resultados obtenidos del desarrollo e implementación del capítulo 4. Se van a explicar cada una de las pruebas realizadas en cada uno de los diseños y los resultados tras la realización de dichas pruebas, también se englobará en este capítulo la valoración de las prestaciones durante la implementación y por último un apartado acerca del esfuerzo realizado como diseñador en este proyecto.

5.1. Pruebas

El desarrollo e implementación del proyecto se ha dividido en dos partes, la primera de ellas consistente en el desarrollo del código del IP core en C utilizando la herramienta de Xilinx Vivado HLS, mientras que por el otro lado se tenía el segundo proceso, el cual consistía en la alteración del diseño de referencia para añadir el diseño en C desarrollado en la primera parte. Para cada una de estas partes se han realizado pruebas distintas a la hora de verificar su correcto funcionamiento y el cumplimiento de las funcionalidades objetivo, por ello esta sección se va a dividir en dos, al igual que la implementación del proyecto, mostrando por separado las pruebas realizadas para cada una de las partes.

5.1.1. Pruebas realizadas sobre el código del IP core

Realizando un recordatorio de lo que hemos visto en la sección 4.2, el IP core tiene como función principal alterar los colores del streaming que se le pasa, esto lo realiza mediante la función **filter_simple_wrapper**, la cual llama a su vez a otras dos funciones distintas. Nuestro objetivo es comprobar utilizando una imagen a modo de streaming, si todos estos pasos son realizados correctamente y obtenemos los resultados esperados.

Una vez se ha finalizado la realización del código del nuevo bloque que se quiere añadir, se ha creado un banco de pruebas o testbench para comprobar su correcto funcionamiento. Esto ha consistido en un código añadido al proyecto de Vivado HLS, el cual se ha ejecutado al realizar la simulación

en C. La funcionalidad de dicho código es dar valores y probar el código principal del IP core, para ello se llamará a la función principal del IP core, pasándole una imagen de nuestra elección y un valor para el nuevo parámetro creado que decidirá que filtro de color se le realiza a la imagen.

Vamos a ir explicando por líneas el proceso que sigue nuestro banco de pruebas o testbench. En primer lugar realizaremos la declaración de las dos funciones **image_to_stream** y **stream_to_image**, tal y como se ve en el código apartado 5.1.1, estas pasarán la imagen que se utiliza de ejemplo a forma streaming y viceversa, esto permite convertir la imagen de ejemplo en un streaming, pasarlo por la función filtro, y volver a pasar el streaming obtenido a la salida de la función de vuelta a una imagen. Finalizar observando que la salida coincida con lo esperado.

Tras el código de las declaraciones se tiene la función main dividida en el código apartado 5.1.1 y el código apartado 5.1.1. El primero inicializa los contadores, le asigna un valor a una variable que luego se pasará al parámetro creado en nuestro filtro, esto se hace para probar los distintos valores que puede tomar ese parámetro, y entonces va recorriendo la imagen línea a línea, convirtiendo a streaming los píxeles dependiendo del parámetro inicial seleccionado. Continúa pasando ese streaming a nuestra función principal **filter_simple_wrapper**.

Código 5.1: En este código se ve la declaración de la imagen de entrada y la de salida junto a las funciones iniciales necesarias para nuestras pruebas con nuestro banco de pruebas.

```
#include "filter_simple.hpp"
#include "kernel_filter.hpp"
#include "bitmap.h"
//Testbench images
#define INPUT_IMAGE "fox.bmp"
#define OUTPUT_IMAGE "fox_output_filter.bmp"
vector <vector <Pixel> > bmp;
void image_to_stream(string filename, stream<video_stream> &stream_in);
void stream_to_image(string filename, stream<video_stream>& stream_out);
```

Pasar esa variable como parámetro permite seleccionar el filtro de color que se desea, o incluso, si no se desea filtrar la imagen al completo, tan solo actuar sobre la mitad de ella, o una cuarta parte, se llega a producir un efecto visualmente muy llamativo en la imagen resultante.

Una vez se ha finalizado la función, se escribe el streaming en la variable imagen de salida y se manda a la función que lo transformará en una imagen, **stream_to_image**, obteniendo el resultado de la prueba.

Código 5.2: En este código tenemos la función principal del testbench, en ella llamaremos a las funciones que pasan de imagen a streaming y viceversa mientras recorremos los píxeles de la imagen de prueba.

```
int main() {
    video_stream currentPixel;
    // Create input and output streams
    stream<video_stream> stream_in("stream_in");
    stream<video_stream> stream_out("stream_out");
    stream<video_stream> stream_out_aux("stream_out_aux");
    int countWidth = -1;
    int countHeight = -1;
    int countUser = -1;
    ap_uint<32> sel = 9;
    // Read Image and transform in a Pixel Stream
    image_to_stream(INPUT_IMAGE, stream_in);
    /* Iterate for every pixel in the image */
    while(!stream_in.empty()){
    filter_simple_wrapper(stream_in, stream_out_aux, sel);
    };
```

Una vez se ha visto la función principal del banco de pruebas, se va a ver que se realiza en el interior de cada una de las funciones a las que llama. La primera de ellas es la función **image_to_stream**. La segunda de ellas es la función **stream_to_image**, donde se leerá un streaming de entrada del cual se irán cogiendo los píxeles de su interior, obteniéndose sus coeficientes de color y almacenándose en una variable que mandaremos como imagen de salida.

Se puede encontrar el código más detallado de las funciones **image_to_stream** y **stream_to_image** en la sección A del apéndice.

Una vez se ha visto cuál es el objetivo del banco de pruebas que se ha diseñado, se va a mostrar cuáles son los resultados arrojados ante los distintos valores dados al parámetro, la funcionalidad para cada uno de los valores que se le da a este parámetro viene dada por el código apartado 5.1.1 que se tiene en el fichero .hpp.

Código 5.3: En este código tenemos la función principal del testbench, en ella llamaremos a las funciones que pasan de imagen a streaming y viceversa mientras recorremos los píxeles de la imagen de prueba.

```
    /*Check if the output stream has the proper signalization, otherwise report an error*/
    countWidth = -1;
    countHeight = -1;
    countUser = -1;
    while(!stream_out_aux.empty()){
        stream_out_aux.read(currentPixel);
        stream_out.write(currentPixel);
        /*Check for user [Start of a Frame (SoF)]*/
        if(currentPixel.user){
            countUser++;
            countHeight++;
            countWidth = 0;
        }
        /*Check for last [End of Line (EoL)]*/
        if(currentPixel.last){
            countHeight++;
            countWidth = 0;
        }
        else
            countWidth++;
        /*Verify if user was "on" for the first */
        if(countUser != 0){
            if (countUser == -1)
                std::cerr << "There is NOT user in the output STREAM check your logic" << std::endl;
            else
                std::cerr << "User has been on multiple times (" << std::dec << countUser << ") in the
                    output STREAM check your logic" << std::endl;
            return -1;
        }
        if (countWidth >= IMAGE_WIDTH){
            std::cerr << "Row (" << countHeight << ") is " << countWidth+1 << " pixels wide which
                are more than the maximum (" << IMAGE_WIDTH << ")" << std::endl;
            return -2;
        }
        if (countHeight > IMAGE_HEIGHT){
            std::cerr << "Image is " << countHeight << " pixels tall which are more than the maximum
                (" << IMAGE_HEIGHT << ")" << std::endl;
            return -3;
        }
    }
    stream_to_image(OUTPUT_IMAGE, stream_out);
    return 0;
}
```

Código 5.4: En este código mostramos los posibles valores del parámetro definido y la función de cada uno, está ubicado en el fichero .hpp para mayor comodidad.

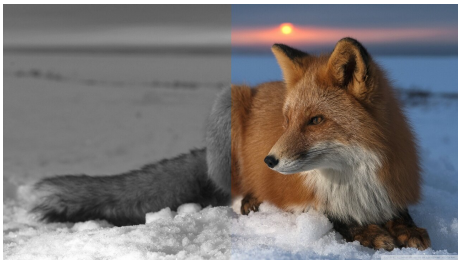
```
//Cases my param
#define IMAGE_ORIGINAL 0 // Case 0 path through
#define IMAGE_GRAY 1 // Case 1 Image gray
#define IMAGE_HOR_GRAY 2 // Case 2 Horizontally divided gray image
#define IMAGE_VER_GRAY 3 // Case 3 Vertically divided gray image
#define IMAGE_DIV_GRAY 4 // Case 4 Vertically and horizontally divided gray image
#define IMAGE_RED 5 // Case 5 Red Image
#define IMAGE_BLUE 6 // Case 6 Blue Image
#define IMAGE_GREEN 7 // Case 7 Green Image
```

Los resultados obtenidos son los que se muestran en las figuras 5.1 y 5.2. Al estar las imágenes de salida en formato .bmp, el cual overleaf no soporta, se ha utilizado un conversor online para transformarlas a .jpg y poder mostrarlas en este documento, este conversor en ningún momento altera el contenido de las imágenes, sino tan solo su formato, podemos acceder a él pulsando aquí.

(a) Imagen resultante case 0

(b) Imagen resultante case 1

(c) Imagen resultante case 2



(d) Imagen resultante case 3

Figura 5.1: Figuras resultantes de las pruebas de asignar los valores 0, 1, 2 y 3 al parámetro, para el valor 0 obtenemos la imagen original, para el 1 la imagen en escala de grises, para el 2 una imagen partida horizontalmente mitad original mitad escala de grises, y para el valor 3 vemos la imagen partida verticalmente mitad orignal mitad en escala de grises. Figuras propias

(a) Imagen resultante case 4

(b) Imagen resultante case 5

(c) Imagen resultante case 6

(d) Imagen resultante case 7

(e) Imagen resultante case else

Figura 5.2: Figuras resultantes de las pruebas de asignar los valores 4, 5, 6, 7 y els al parámetro creado, para el valor 4 obtenemos la imagen partida horizontal y verticalmente alternando entre original y grises, para el 5 obtenemos la imagen filtrada en color rojo, para el 6 la imagen filtrada para el color azul, y para el 7 resulta en la imagen filtrada para el color verde, para un valor cualquiera fuera del rango del case entre 0 y 7, la imagen debe resultar en una combinación aleatoria de sus propios colores. Figuras propias

5.1.2. Pruebas realizadas sobre el diseño final en la placa

Una vez probado el IP core se ha pasado a la sección 4.3, donde se han realizado las modificaciones en el overlay hardware, ahora se van a realizar las pruebas para verificar los resultados de estas modificaciones. Estas pruebas han consistido en conectar al puerto HDMI de entrada un reproductor de video, mientras que al HDMI de salida se le ha conectado una pantalla, así se verá por dicha pantalla la señal de video filtrada en color por el IP core.

Como se ha visto en la figura 4.10 de la sección de la implementación, ya se ha cargado en un notebook de Jupyter el overlay y las librerías con las funciones necesarias para guardarlas en una variable llamada base y poder acceder a todas las funciones ya existentes e implementadas directamente desde esa variable.

El objetivo ahora es la implementación de un código python en el notebook de jupyter, justo posterior a la carga del overlay, primero se realizará un primer código que simplemente conectará el HDMI de entrada al de salida, para verificar que la configuración, sin alterar ningún parámetro, funciona correctamente, este código se puede ver en la figura 5.3, y lo que se obtiene con él por pantalla en la figura 5.4. Sin embargo, se busca probar todas las opciones alterando el parámetro añadido por el IP core, esto se hace mediante el código que se puede apreciar en la figura 5.5, con él se observará que, efectivamente, el video que se obtiene a la salida sale filtrado por color en función del parámetro, tal y como muestra la figura 5.6. Para observar que todos los valores del parámetro cumplen su función, se ha creado un bucle que va probando valores del parámetro desde el 0 hasta el 9 con un tiempo entre modificaciones de 2 segundos, esto hará que se observe en la pantalla de salida, la imagen de video de entrada cambiando de color cada 2 segundos según el filtro correspondiente, tal y como se tenía planificado.

```
In [7]: hdmi_in.configure()
        hdmi_out.configure(hdmi_in.mode)

        hdmi_in.start()
        hdmi_out.start()

        hdmi_in.tie(hdmi_out)

        Frequency: 74250000

In [5]: hdmi_out.close()
        hdmi_in.close()
```

Figura 5.3: Figura del código python que inicializa la configuración de los puertos HDMIs y los conecta entre ellos para comprobar su funcionamiento de base. Figura propia

Aunque no se muestren en las figuras también se han comprobado la correcta variación del parámetro entre los dos valores restantes, aunque por cuestiones de velocidad del video no se han podido muestrear.

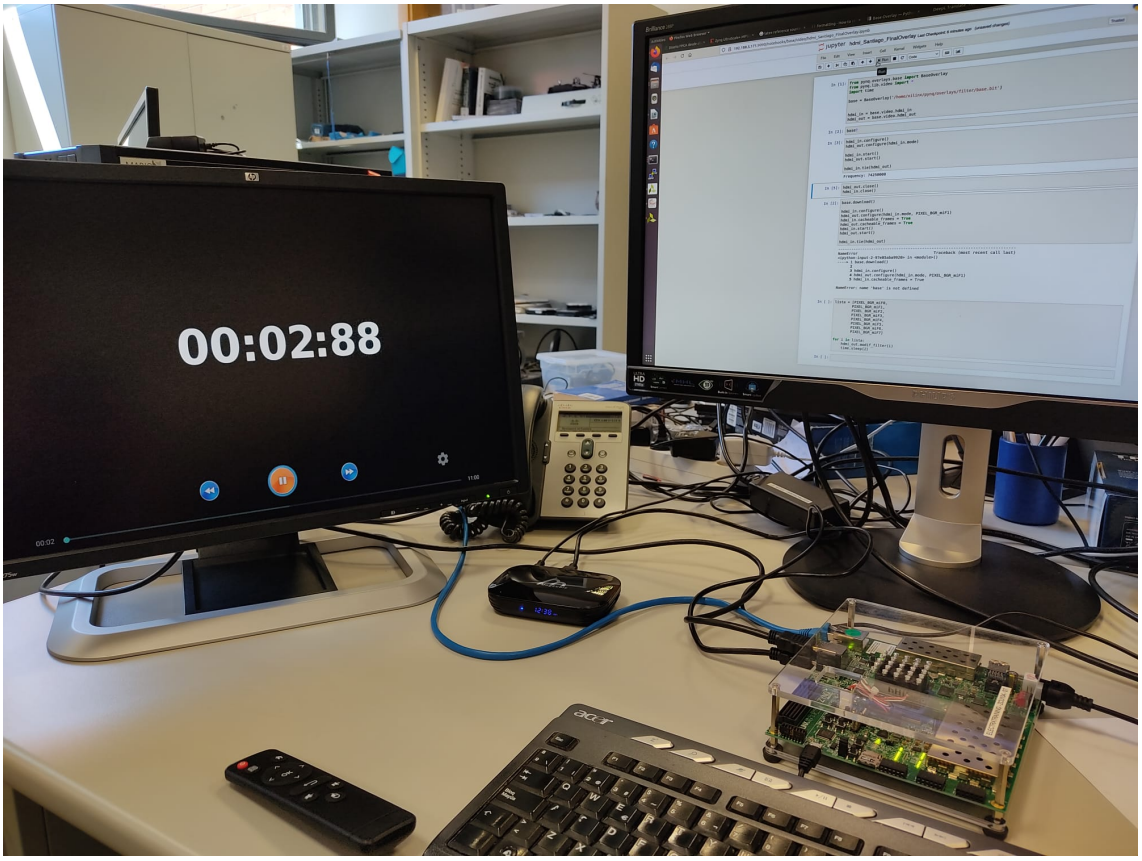


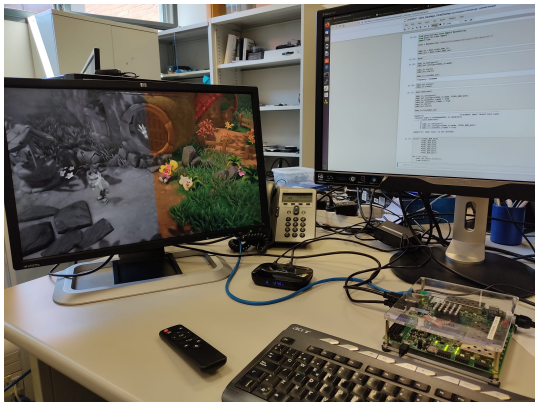
Figura 5.4: Figura mostrando la imagen de la pantalla de salida al solo conectar el HDMI de entrada al de salida como prueba. Figura propia

```
In [ ]: lista = [PIXEL_BGR_miF0,
                PIXEL_BGR_miF1,
                PIXEL_BGR_miF2,
                PIXEL_BGR_miF3,
                PIXEL_BGR_miF4,
                PIXEL_BGR_miF5,
                PIXEL_BGR_miF6,
                PIXEL_BGR_miF7]

        for i in lista:
            hdmi_out.modif_filter(i)
            time.sleep(2)

In [ ]:
```

Figura 5.5: Figura del código python creado en el notebook de jupyter que prueba todas las posibilidades del parámetro de nuestro IP core. Figura propia

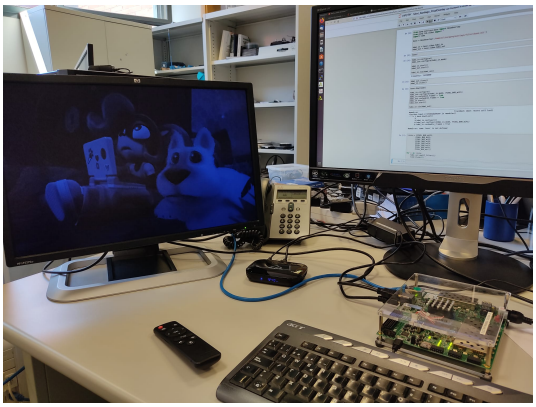
(a) Imagen final resultante para 0

(b) Imagen final resultante para 2

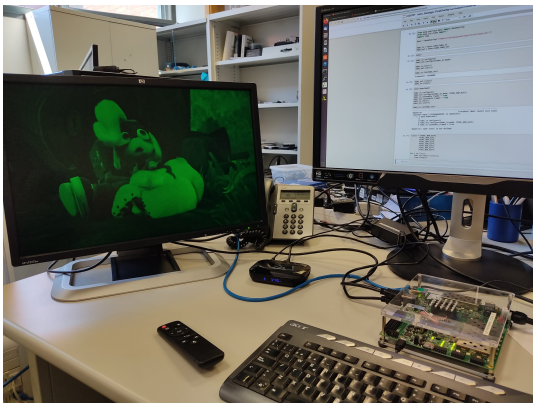


(c) Imagen final resultante para 3

(d) Imagen final resultante para 5



(e) Imagen final resultante para 6



(f) Imagen final resultante para 7

Figura 5.6: Figuras resultantes de la prueba final al variar el parámetro por cada uno de sus valores. Figuras propias

5.2. Evaluación de prestaciones

En esta sección se va a evaluar el rendimiento del proyecto completo, y el rendimiento de forma individual de la función añadida, a fin de comparar los recursos utilizados de forma global con los recursos utilizados exclusivamente por la función añadida. Con estos datos se podrá obtener el porcentaje aproximado de recursos al añadir una función adicional y comprobar si es viable dentro del proyecto.

También se verá qué elementos del proyecto son los que más consumen y que tipo de recursos son los más importantes, a su vez nos interesa saber si cumple con los requisitos de tiempos de procesamiento impuestos por los propios diseñadores. Otro elemento a comprobar es la frecuencia a la que se está trabajando, este proyecto posee distintos relojes los cuales funcionan a frecuencias distintas entre ellos y cada bloque que conforma el proyecto está conectado a uno de ellos, siguiendo la conexión de reloj se puede encontrar cuál afecta a nuestro bloque de video y más concretamente a la función que hemos incorporado.

Tanto para observar los recursos utilizados como los relojes del proyecto, así como los tiempos utilizados, se va al proyecto en Vivado donde añadimos el IP core, y entre las opciones de implementación, al abrir el diseño implementado se tiene una opción de reporte de utilización, tras pulsarla Vivado muestra la ventana que se ve en la imagen de la figura 5.7.

Tal y como se puede apreciar en dicha figura, la cantidad de recursos utilizada por la función añadida es muy pequeña comparada con el total de recursos utilizados por el proyecto, los recursos que más consume son CLB, LUTs y registros en CLB, sin embargo para cada uno de ellos se obtiene una utilización de un 0,841 % y un 0,926 % respectivamente en comparación con el total de cada uno de esos recursos utilizados por el proyecto global, si se observa el número de bloques RAM que utiliza, se verá que no utiliza ninguno, a pesar de que el total del proyecto utilice 61. Como se puede ver es una cantidad muy baja de recursos utilizados en general, despreciable prácticamente, lo que muestra una vez más que realizar esta clase de modificaciones no altera su eficiencia y a su vez permite añadir mucha funcionalidad, prueba de su gran versatilidad y eficiencia.

Name	CLB LUTs (230400)	CLB Registers (460800)	CARRY8 (28800)	F7 Muxes (115200)	F8 Muxes (57600)	CLB (28800)	LUT as Logic (230400)	LUT as Memory (101760)	Block RAM Tile (312)	URAM (96)
N base_wrapper	51338	68584	860	694	52	11815	46027	5311	61	2
base_i (base)	51338	68584	860	694	52	11815	46027	5311	61	2
address_remap_0 (base_address_remap_0_0)	0	0	0	0	0	0	0	0	0	0
axi_intc_0 (base_axi_intc_0_0)	140	147	0	0	0	36	140	0	0	0
axi_interconnect (base_axi_interconnect_0)	14857	19532	20	24	0	3459	13241	1616	0	0
axi_interconnect_0 (base_axi_interconnect_0_(	666	759	0	0	0	173	644	22	0	0
axi_mem_intercon (base_axi_mem_intercon_0)	0	0	0	0	0	0	0	0	0	0
axi_mem_intercon_1 (base_axi_mem_intercon_	0	0	0	0	0	0	0	0	0	0
fmch_axi_iic (base_fmch_axi_iic_0)	392	385	4	8	4	84	374	18	0	0
gpio_btns (base_gpio_btns_0)	72	109	0	0	0	20	72	0	0	0
gpio_leds (base_gpio_leds_0)	36	52	0	0	0	10	36	0	0	0
gpio_sws (base_gpio_sws_0)	72	109	0	0	0	22	72	0	0	0
iop_pmod0 (iop_pmod0_imp_GBIRKK)	3380	3865	67	119	4	795	3223	157	16	0
iop_pmod1 (iop_pmod1_imp_24UQP0)	3384	3865	67	119	4	787	3227	157	16	0
mb_iop_pmod0_intr_ack (base_mb_iop_pmod0_	0	0	0	0	0	0	0	0	0	0
mb_iop_pmod0_reset (base_mb_iop_pmod0_re	0	0	0	0	0	0	0	0	0	0
mb_iop_pmod1_intr_ack (base_mb_iop_pmod1_	0	0	0	0	0	0	0	0	0	0
mb_iop_pmod1_reset (base_mb_iop_pmod1_re	0	0	0	0	0	0	0	0	0	0
mdm (base_mdm_0)	90	112	0	0	0	23	83	7	0	0
pmod0_buf (base_pmod0_buf_0)	0	0	0	0	0	0	0	0	0	0
pmod1_buf (base_pmod1_buf_0)	0	0	0	0	0	0	0	0	0	0
proc_sys_reset_0 (base_proc_sys_reset_0_0)	17	39	0	0	0	5	16	1	0	0
proc_sys_reset_1 (base_proc_sys_reset_1_0)	14	39	0	0	0	14	13	1	0	0
ps_e_0 (base_ps_e_0_0)	0	0	0	0	0	0	0	0	0	0
reset_control (base_reset_control_0)	58	160	0	0	0	31	58	0	0	0
shutdown_HP0_FPD (base_shutdown_HP0_FPD	863	1900	0	0	0	235	367	496	0	0
shutdown_HP2_FPD (base_shutdown_HP2_FPD	682	1396	0	0	0	195	322	360	0	0
shutdown_LPD (base_shutdown_LPD_0)	895	1915	0	0	0	294	399	496	0	0
vcc_const (base_vcc_const_0)	0	0	0	0	0	0	0	0	0	0
video (video_imp_1KRF0RE)	25740	34200	702	424	40	6247	23760	1980	29	2
axi_vdma (base_axi_vdma_0)	2713	4238	35	1	0	604	2533	180	15	2
const_gnd (base_const_gnd_0)	0	0	0	0	0	0	0	0	0	0
hdmi_in (hdmi_in_imp_1J5I8O)	9198	10708	66	170	16	2171	8452	746	2.5	0
hdmi_out (hdmi_out_imp_1RSCXJD)	6937	10243	66	154	0	1783	5883	1054	11.5	0
color_convert (base_color_convert_1)	388	627	12	0	0	108	362	26	0	0
filter_simple_wrapper_0 (base_filter_simpl	432	645	16	0	0	128	432	0	0	0
frontend (base_frontend_1)	5682	8226	38	154	0	1448	4654	1028	11.5	0
pixel_reorder (base_pixel_reorder_1)	0	0	0	0	0	0	0	0	0	0
pixel_unpack (base_pixel_unpack_0)	405	640	0	0	0	110	405	0	0	0
tx_video_axis_reg_slice (base_tx_video_ax	30	105	0	0	0	12	30	0	0	0
phy (phy_imp_12V6S53)	6892	9011	535	99	24	1791	6892	0	0	0
xlconcat0 (base_xlconcat0_0)	0	0	0	0	0	0	0	0	0	0
xlconcat_0 (base_xlconcat_0_0)	0	0	0	0	0	0	0	0	0	0
xlconcat_1 (base_xlconcat_1_0)	0	0	0	0	0	0	0	0	0	0
xlslice_0 (base_xlslice_0_0)	0	0	0	0	0	0	0	0	0	0
xlslice_1 (base_xlslice_1_0)	0	0	0	0	0	0	0	0	0	0

Figura 5.7: Figura del resultado del reporte de utilización obtenido de vivado, aquí podemos ver cada uno de los recursos hardware utilizado por cada uno de los elementos que conforman el proyecto. Figura propia

Siguiendo, se van a observar cuáles son los elementos de la jerarquía del proyecto que más cantidad de recursos consumen de este mismo, en la figura 5.7 que es la analizada previamente, se puede apreciar que el bloque de video, y en concreto los bloques correspondientes a los HDMIs de entrada, de salida y el bloque phy, que constituyen el bloque de video, consumen una buena cantidad de recursos, en total el bloque de video utiliza un 50,138 % del total de CLB LUTs utilizadas, mientras que de registros utiliza un 49,865 %. Sin embargo lo que con diferencia utiliza más recursos en el proyecto, son las interconexiones entre los elementos del mismo, tan solo ellas ya suponen un 28,939 % de las CLB LUTs del proyecto, aunque el bloque de video utilice más estos recursos se distribuyen por los bloques que lo conforman mientras que la cantidad utilizada por las interconexiones no es nada despreciable para ser tan solo interconexiones entre elementos.

Observando la figura 5.8, se ve que cumple con los tiempos mínimos estipulados por los diseñadores para su correcto funcionamiento y también se analizan los requisitos de tiempo puestos por el hardware de la placa y el resto del diseño.

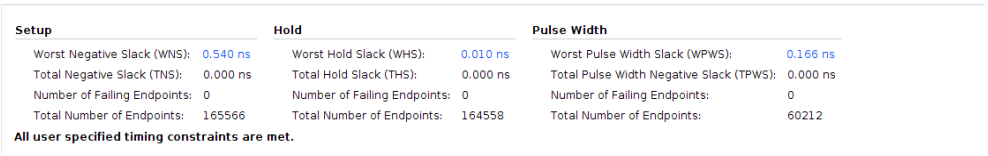


Figura 5.8: Figura del resultado del reporte de tiempos utilizado por el reloj del proyecto que influye sobre los bloques de interés, sobre los que hemos trabajado en el proyecto. Figura propia

Dentro del proyecto hay cuatro relojes globales a los que todos los elementos están conectados, a su vez hay ciertos relojes exclusivos de otros elementos que no se van a ver en este trabajo. Siguiendo las interconexiones del bloque añadido se ha visto que el reloj que se utiliza es el reloj número 1, en la figura 5.8 lo que se ve es el reporte de tiempos del reloj 1, el que afecta a nuestra función, en la figura 5.9.

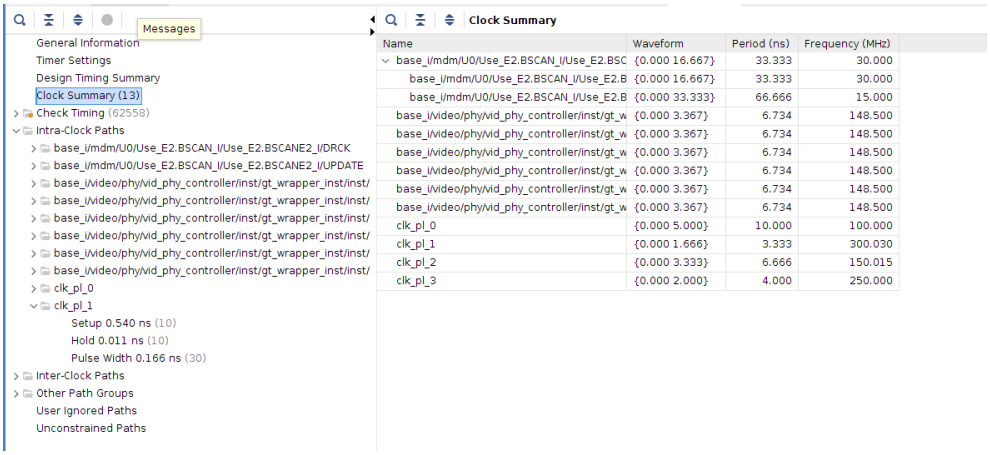


Figura 5.9: Figura del listado de relojes que conforman el proyecto junto a sus frecuencias de trabajo. Figura propia

Como se ve cada reloj tiene su propia frecuencia, en concreto el nuestro, que es el clk_pl_1, está trabajando a una frecuencia de 300 MHz.

5.3. Esfuerzo de diseño

En este apartado se quiere tratar el esfuerzo que, como diseñador, ha conllevado el desarrollo de este proyecto, desde su etapa de investigación y familiarización inicial hasta la conclusión de este documento. Para ello se ha realizado un diagrama de Gantt que se puede ver en la figura 5.10.

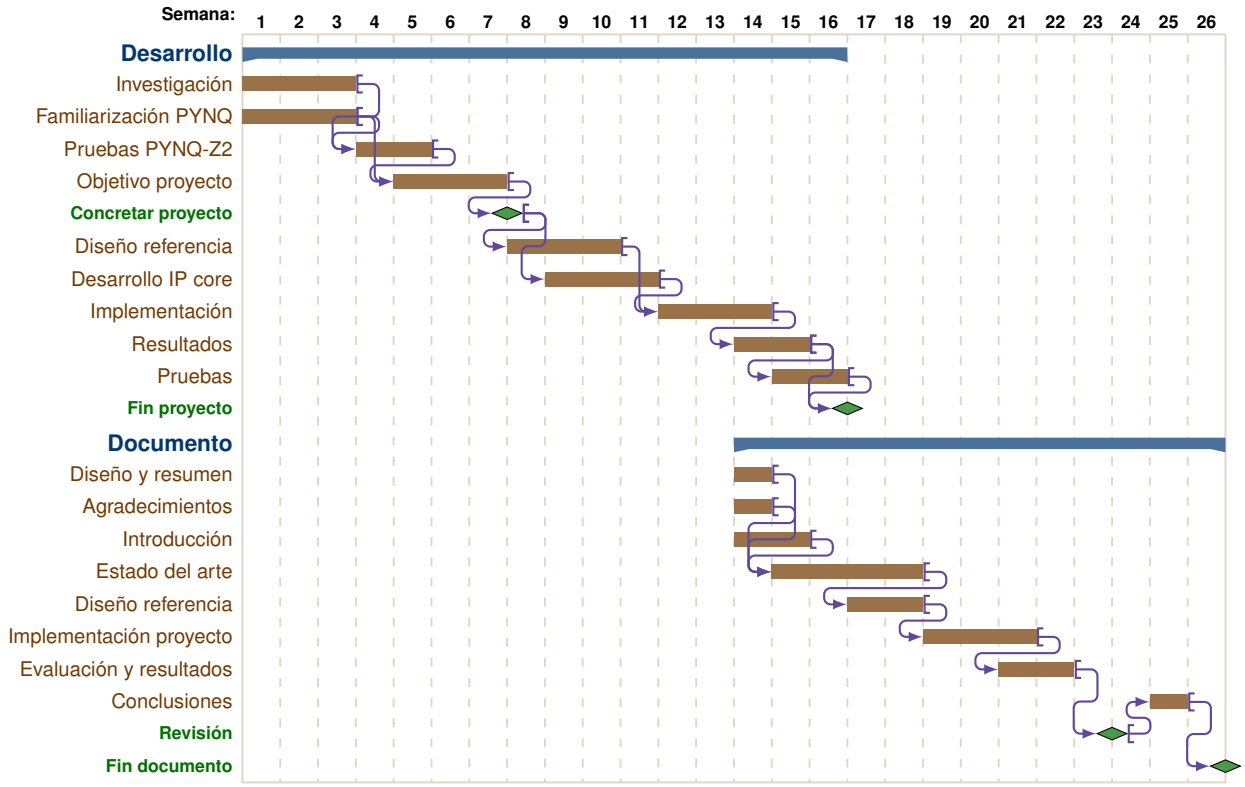


Figura 5.10: Diagrama de Gantt de este TFG, en él se incluye desde la investigación inicial del proyecto hasta la finalización de la memoria.

Este trabajo ha consistido en dos partes, las cuales desde el punto de vista como diseñador, se ha de remarcar la parte del desarrollo e implementación del proyecto, dejándose la parte de la escritura del documento esquematizada en el diagrama de Gantt.

Dentro del proyecto ha sido una labor continua de investigación a la par que de desarrollo, de familiarización con la estructura, el diseño y las distintas plataformas para las que está implementado. Tras unas semanas iniciales de investigación y familiarización con el proyecto realizando algunos de los proyectos de prueba de la comunidad, los cuales utilizando placas de menor rendimiento y complejidad tuvieron el objetivo de ayudar a comprender mejor la base sobre la que se asentaba todo el diseño.

Tras esto se ha pasado a concretar exactamente cuál iba a ser el objetivo de mi trabajo, el proyecto de PYNQ ofrece amplias posibilidades y multitud de funcionalidades que se podrían abordar de forma individual como proyecto propio. Esto de por sí nos dice mucho sobre la magnitud del diseño global. Por ello tras amplia investigación y varias reuniones se decidió optar por un enfoque basado en la funcionalidad del procesamiento de video de este proyecto.

Tras verse que las placas de bajo rendimiento tales como la PYNQ-Z1 y la PYNQ-Z2 no tenían la suficiente capacidad de procesamiento de video como para tener la posibilidad de analizar de forma

satisfactoria un filtrado de video, se decidió utilizar una placa de mayor rendimiento, ésta es la ZCU104, sobre la cual ya se ha hablado múltiples veces en este documento.

Una vez concretado el proyecto el siguiente paso fue hacer un análisis en profundidad del diseño de referencia, ya que se iba a modificar, debíamos conocer y comprender bien todo el flujo de diseño que sigue, para ello se eligió uno de los bloques de procesamiento de video que posee el diseño y se realizó la modificación, a fin de seguir paso a paso el flujo de diseño que dicha modificación iba a producir, con el objetivo de usarse de referencia para la posterior alteración del diseño, este bloque que se alteró fue el color converter, un IP core perteneciente al bloque de video del diseño cuya funcionalidad es la de analizar los colores de las imágenes entrantes por los puertos HDMI .

Una vez conocido a la perfección el flujo completo se pasó a diseñar el IP core desarrolladose en C mediante Vivado HLS, una vez finalizado y probado se pasó a añadirlo al diseño de referencia, este fue el paso que más problemas conllevó, ya que añadir un bloque nuevo al diseño sigue ciertas pautas que no son necesarias en la modificación de uno ya existente, por lo que esta fase tomó más tiempo del planeado anteriormente, se utilizó múltiples tutoriales y guías acerca del proceso de añadir un bloque nuevo de cero al diseño, a pesar de que en su mayoría eran bloques mucho más simples que el nuestro, el procedimiento era similar.

Una vez se completó la implementación se inició la fase de pruebas y la valoración de resultados, esta fase fue la más asequible del proyecto, ya que la mayor parte de estas pruebas funcionaron obteniendo los resultados esperados.

6

CONCLUSIONES

Como conclusión a este trabajo se quieren analizar los objetivos del proyecto de PYNQ , así como los objetivos propios de este trabajo, los cuales se establecieron en la sección 1.2.

Tal y como su nombre indica Python Productivity for Zynq busca volver más sencillo el uso de las plataformas de Xilinx, dando a los diseñadores la capacidad de utilizar la lógica programable y los microprocesadores a través del lenguaje de programación de alto nivel Python. Los desarrolladores pueden ahora, con tan solo un mínimo conocimiento acerca de diseño hardware, crear sistemas electrónicos que combinen la versatilidad de un lenguaje de alto nivel con las capacidades de alto rendimiento de este mismo.

Ahora los diseñadores software, con poseer este mínimo conocimiento acerca del hardware podrán diseñar aplicaciones de alto rendimiento que se puedan ejecutar en hardware de forma paralela, que tengan procesamiento de video con un ratio de frames muy elevado, que implementen algoritmos de aceleración del hardware, que hagan procesamiento en tiempo real de la señal, que tengan un ancho de banda de entrada y salida muy grande y con un control de latencia muy bajo.

Viéndose todas estas posibilidades y habiéndose demostrado a lo largo de todo este trabajo que son factibles, solo cabe decir que el proyecto cumple con creces los objetivos y expectativas que lo han llevado a su desarrollo, y que por ello posee un gran atractivo para todas las personas que trabajan en el mundo tanto del codiseño hardware como software.

Se tratarán ahora los objetivos de este trabajo descritos al principio del documento. El proyecto que se ha implementado en este trabajo ha consistido en, con cierto nivel de investigación y conocimientos básicos acerca del hardware a utilizar, ser capaces de alterar este mismo y de controlarlo desde un notebook de Jupyter utilizando el lenguaje de alto nivel Python.

Los cambios realizados en el diseño base no han alterado el mismo de forma abrupta en lo absoluto, como se ha visto en la sección de prestaciones 5.2, quedando así demostrado que, gracias a este flujo de diseño, no son necesarias modificaciones excesivas para poder controlar el hardware desde un lenguaje de programación en el software.

Así mismo se ha completado la modificación y comprobado su correcto funcionamiento en la sec-

ción 5.1, probándose en esta misma todas las funcionalidades añadidas, lo cual de por sí verifica que se ha cumplido el objetivo principal.

El último objetivo de este trabajo consistía en realizar un análisis en profundidad acerca de las posibilidades que ofrece este proyecto en la actualidad, este mismo objetivo coincide con uno de los objetivos del proyecto en sí, y por tanto de sus desarrolladores, y su cumplimiento se ha comentado al principio del capítulo 6.

6.1. Trabajos futuros

Actualmente PYNQ es un proyecto que todavía sigue en desarrollo, la empresa que lo creó, Xilinx, continúa sacando nuevas versiones del mismo y actualizándolo de forma periódica, es un proyecto en el que aquellos que forman parte del programa universitario de Xilinx han estado y están trabajando, llegando a incluir en algunos casos alumnos de la Escuela Politécnica de la Universidad Autónoma de Madrid.

La última actualización del código fuente donde se encuentra el diseño de referencia, es a fecha actual, se está ante un proyecto en pleno crecimiento y desarrollo, proyecto que posee un gran potencial. Ya que las placas y sistemas electrónicos de Xilinx son actualizadas y mejoradas de forma constante, también sus codiseños tanto hardware como software, lo son, adaptándose a estas mejoras y cambios a la vez que desarrollando nuevas e increíbles funcionalidades.

Bibliografía

[1] R. Alonso, “En hardware se usa mucho el término soc pero, ¿sabes qué es?,” *Hardzone*, Julio 2020. (Leer).

[2] S. Harish, “SOC architecture and design.” (Leer), Mayo 2015.

[3] W. G. Wong, “Reconfigurable socs,” *Electronic Design*, Octubre 2000. (Leer).

[4] G. Sutter, “Introduction to embedded sistems and design flow in xilinx soc,” (Descargar).

[5] “Vivado Design Suite User Guide. Embedded Processor Hardware Design.” (Leer o descargar), Junio 2020.

[6] “Vitis Unified Software Platform Documentation. Embedded Software Development.” (Leer o descargar), Junio 2020.

[7] “PYNQ Introduction.” (Leer), 2018.

[8] “PYNQ Overlays.” (Leer), 2018.

[9] “Python Environment.” (Leer), 2018.

[10] “Jupyter Notebooks.” (Leer), 2018.

[11] “What is PYNQ?.” (Leer).

[12] Xilinx, “Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit.” (Leer), 2021.

[13] C. C. Franco, “Desarrollo de aceleradores hardware con técnicas de High-Level Synthesis: exploración de alternativas con paralelismo explícito y ejecución escalable.” (Leer o descargar), Septiembre 2016.

[14] D. Payne, “High-level synthesis and open source software algorithms,” Julio 2020. (Leer).

[15] C. McCabe and Contributors, “Xilinx/PYNQ.” (Leer o descargar), Junio 2021.

[16] “Base Overlay. ZCU104 Block Diagram.” (Leer), 2018.

Definiciones

bits Bit es la abreviación de Binary Digit y es la menor unidad de información de una computadora, toda la información de una computadora es medida y codificada en bits, es una medida de almacenamiento de datos.

overflow EL overflow se refiere al desbordamiento de buffer, esto es cuando un programa no gestiona correctamente la memoria que utiliza con la cantidad de datos que maneja y parte de estos datos se pierden al no caber en el volumen de memoria que está utilizando.

streaming Streaming refiere a un flujo constante de datos tanto de entrada como de salida, enviados o recibidos, que permite su procesamiento incluso cuando aun no se ha recibido al completo.

Acrónimos

ADAS Advanced Driver Assistance Systems.

API Application Programming Interface.

APSoC Array Processor (AP) on a chip (SoC).

AR Augmented Reality.

FPGA Field Programmable Gate Array.

HDL Hardware Description Language.

HDMI High-Definition Multimedia Interface.

HLS High-Level Synthesis.

IDE Integrated Development Environment.

IP Internet Protocol.

IPython Interactive Python.

MPEG-1 Moving Picture Experts Group 1.

MPEG-2 Moving Picture Experts Group 2.

PL Programmable Logic.

PS Processing System.

PYNQ Python Productivity for Zynq.

RFSoC Radio Frequency Simulation Operations Center.

RTL Register Tranfer Language.

SoC System on a Chip.

TCL Tool Command Language.

TFG Trabajo de fin de grado.

VPN Virtual Private Network.

Apéndices

A

Desarrollo y pruebas del IP core

En este capítulo del anexo vamos a incluir el código detallado de las funciones que conforman partes del trabajo. Este código no es necesario para la correcta comprensión del diseño del IP core en la sección 4.2, ni de las pruebas realizadas sobre este mismo core en la sección 5.1, ya que se encuentran detallados dentro de esas mismas secciones, pero están aquí incluidos como información más detallada y adicional a dichas secciones.

A.0.1. Diseño IP core

Código A.1: En esta figura se presenta el código correspondiente a las declaraciones de la funcion add_gray.

```
void add_gray(stream<video_stream>& stream_in_48, stream<internal_stream>& stream_internal64) {
#pragma HLS pipeline II=1
    /*Declare unsigned fixed point variables: 9-bit in total with 1-bit for the integer part*/
    const ap_ufixed<9,1> redConst = 0.2126;
    const ap_ufixed<9,1> greenConst = 0.7152;
    const ap_ufixed<9,1> blueConst = 0.0722;
    /*Auxiliary stateless variables for processing*/
    video_stream    input_pixel;
    internal_stream    output_pixel;
    ap_uint<8>     red0, red1;
    ap_uint<8>     blue0, blue1;
    ap_uint<8>     green0, green1;
    ap_uint<9>    luminance0, luminance1 = 0;
    if (!stream_in_48.empty()){
        stream_in_48.read(input_pixel);
        output_pixel.user = input_pixel.user;
        output_pixel.last = input_pixel.last;

        red0 = input_pixel.pixels.p0_c1;
        blue0 = input_pixel.pixels.p0_c2;
        green0 = input_pixel.pixels.p0_c3;

        red1 = input_pixel.pixels.p1_c1;
        blue1 = input_pixel.pixels.p1_c2;
        green1 = input_pixel.pixels.p1_c3;
```

Código A.2: En esta figura se presenta el código correspondiente a la funcion add_gray que pasa los bits que recibe en forma RGB a escala de grises.

```
    /*Dissects color channels*/
    //(red,green,blue) = input_pixel.data;
    /*Convert from RGB to grayscale*/
    luminance0 = redConst *red0 + greenConst *green0 + blueConst *blue0;
    luminance1 = redConst *red1 + greenConst *green1 + blueConst *blue1;
    /*Check overflow, it shouldn't be necessary but just in case*/
    if (luminance0 > 255)
        output_pixel.pixels.p0_gr = 255;
    else
        output_pixel.pixels.p0_gr = luminance0; // the extra bit is shaved-off by the logic
    /*Check overflow, it shouldn't be necessary but just in case*/
    if (luminance1 > 255)
        output_pixel.pixels.p1_gr = 255;
    else
        output_pixel.pixels.p1_gr = luminance1; // the extra bit is shaved-off by the logic

    /*Populate original data and signalization*/
    output_pixel.pixels.pix_RGB = input_pixel.pixels;
    /*Write out stream*/
    stream_internal64.write(output_pixel);
    }
}
```

Código A.3: En esta figura se presenta el código correspondiente a las declaraciones de la funcion filter_simple.

```
void filter_simple( stream<internal_stream>& stream_internal64, stream<video_stream>&
      stream_out_48, ap_uint<32>& my_param) {
#pragma HLS pipeline II=1
    internal_stream    input_pixel;
    video_stream    output_pixel;
    static int rows_counter = 0;  // from 0 to 1023
    static int columns_counter = 0; // from 0 to 2047
    static bool   vertical_half;
    static bool   horizontal_half;
    if (!stream_internal64.empty()){
        stream_internal64.read(input_pixel);
        output_pixel.user = input_pixel.user;
        output_pixel.last = input_pixel.last;
```

Código A.4: En esta figura se presenta la primera mitad del código correspondiente a la funcion filter_simple, que en funcion del valor de un parametro altera los colores de la imagen a nuestra elección.

```
switch (my_param){
        case IMAGE_ORIGINAL:
             output_pixel.pixels = input_pixel.pixels.pix_RGB;
             break;
        case IMAGE_GRAY:
            output_pixel.pixels.p0_c1 = input_pixel.pixels.p0_gr;
            output_pixel.pixels.p0_c2 = input_pixel.pixels.p0_gr;
            output_pixel.pixels.p0_c3 = input_pixel.pixels.p0_gr;
            output_pixel.pixels.p1_c1 = input_pixel.pixels.p1_gr;
            output_pixel.pixels.p1_c2 = input_pixel.pixels.p1_gr;
            output_pixel.pixels.p1_c3 = input_pixel.pixels.p1_gr;
            break;
        case IMAGE_HOR_GRAY:
            if (!horizontal_half){
                output_pixel.pixels.p0_c1 = input_pixel.pixels.p0_gr;
                output_pixel.pixels.p0_c2 = input_pixel.pixels.p0_gr;
                output_pixel.pixels.p0_c3 = input_pixel.pixels.p0_gr;
                output_pixel.pixels.p1_c1 = input_pixel.pixels.p1_gr;
                output_pixel.pixels.p1_c2 = input_pixel.pixels.p1_gr;
                output_pixel.pixels.p1_c3 = input_pixel.pixels.p1_gr;
            }
            else{
                output_pixel.pixels = input_pixel.pixels.pix_RGB;
            }
            break;
        case IMAGE_VER_GRAY:
            if (!vertical_half){
                output_pixel.pixels.p0_c1 = input_pixel.pixels.p0_gr;
                output_pixel.pixels.p0_c2 = input_pixel.pixels.p0_gr;
                output_pixel.pixels.p0_c3 = input_pixel.pixels.p0_gr;
                output_pixel.pixels.p1_c1 = input_pixel.pixels.p1_gr;
                output_pixel.pixels.p1_c2 = input_pixel.pixels.p1_gr;
                output_pixel.pixels.p1_c3 = input_pixel.pixels.p1_gr;
            }
            else{
                output_pixel.pixels = input_pixel.pixels.pix_RGB;
            }
            break;
```

Código A.5: En esta figura se presenta la segunda mitad del código correspondiente a la funcion filter_simple que en funcion del valor de un parametro altera los colores de la imagen a nuestra elección.

```
            case IMAGE_DIV_GRAY:
                if (vertical_half ^ horizontal_half){
                    output_pixel.pixels.p0_c1 = input_pixel.pixels.p0_gr;
                    output_pixel.pixels.p0_c2 = input_pixel.pixels.p0_gr;
                    output_pixel.pixels.p0_c3 = input_pixel.pixels.p0_gr;
                    output_pixel.pixels.p1_c1 = input_pixel.pixels.p1_gr;
                    output_pixel.pixels.p1_c2 = input_pixel.pixels.p1_gr;
                    output_pixel.pixels.p1_c3 = input_pixel.pixels.p1_gr;
                }
                else{
                    output_pixel.pixels = input_pixel.pixels.pix_RGB;
                }
                break;
            case IMAGE_RED:
                output_pixel.pixels.p0_c1 = input_pixel.pixels.pix_RGB.p0_c1;
                output_pixel.pixels.p0_c2 = 0;
                output_pixel.pixels.p0_c3 = 0;
                output_pixel.pixels.p1_c1 = input_pixel.pixels.pix_RGB.p1_c1;
                output_pixel.pixels.p1_c2 = 0;
                output_pixel.pixels.p1_c3 = 0;
                break;
            case IMAGE_BLUE:
                output_pixel.pixels.p0_c1 = 0;
                output_pixel.pixels.p0_c2 = input_pixel.pixels.pix_RGB.p0_c2;
                output_pixel.pixels.p0_c3 = 0;
                output_pixel.pixels.p1_c1 = 0;
                output_pixel.pixels.p1_c2 = input_pixel.pixels.pix_RGB.p1_c2;
                output_pixel.pixels.p1_c3 = 0;
                break;
            case IMAGE_GREEN:
                output_pixel.pixels.p0_c1 = 0;
                output_pixel.pixels.p0_c2 = 0;
                output_pixel.pixels.p0_c3 = input_pixel.pixels.pix_RGB.p0_c3;
                output_pixel.pixels.p1_c1 = 0;
                output_pixel.pixels.p1_c2 = 0;
                output_pixel.pixels.p1_c3 = input_pixel.pixels.pix_RGB.p1_c3;
                break;
            default:
                output_pixel.pixels.p0_c1 = input_pixel.pixels.pix_RGB.p0_c3;
                output_pixel.pixels.p0_c2 = input_pixel.pixels.pix_RGB.p0_c2;
                output_pixel.pixels.p0_c3 = input_pixel.pixels.pix_RGB.p0_c1;
                output_pixel.pixels.p1_c1 = input_pixel.pixels.pix_RGB.p1_c3;
                output_pixel.pixels.p1_c2 = input_pixel.pixels.pix_RGB.p1_c2;
                output_pixel.pixels.p1_c3 = input_pixel.pixels.pix_RGB.p1_c1;
                break;
    }
```

Código A.6: En esta figura se presenta el código correspondiente a la funcion que verifica las señales de fin de linea y de streaming y que va seleccionando los bits de a dos.

```
    /* Verify TUSER and TLAST signals*/
    if (input_pixel.user){  // When a new image starts reset rows and columns counters
        rows_counter  = 0;
        columns_counter = 0;
    }
    else{
        if(input_pixel.last){ // When the last signal is set reset increase rows_counter and reset
            columns_counter
            rows_counter += 1;
            columns_counter = 0;
        }
        else {
            columns_counter +=2;
        }
    }
    vertical_half = (columns_counter > IMAGE_WIDTH/2);
    horizontal_half = (rows_counter > IMAGE_HEIGHT/2);
    /*Write out stream*/
    stream_out_48.write(output_pixel);
    }
}
```

A.0.2. Pruebas IP core

Código A.7: En este código tenemos la función image_to_stream la cual recibe una imagen y un streaming y, dividiendo cada pixel de la imagen en sus colores RGB, los escribe en el streaming que recibe.

```
void image_to_stream(string filename, stream<video_stream>& stream_in_48){
    Bitmap image;
    //vector <vector <Pixel> > bmp;
    Pixel rgb0, rgb1;
    video_stream currentPixel;
    //read a file example.bmp and convert it to a pixel matrix
    image.open(filename);
    bmp = image.toPixelMatrix();
    // Convert from Matrix to AXI4-Stream
    currentPixel.user = 1; //first Pixel of image
    for (int i=0; i < IMAGE_HEIGHT; i++){
        for (int j=0; j < IMAGE_WIDTH; j+=2){
            rgb0 = bmp[i][j];
            rgb1 = bmp[i][j+1];
            currentPixel.last = (j == (IMAGE_WIDTH-2)) ? 1 : 0;
            currentPixel.pixels.p0_c1 = rgb0.red;
            currentPixel.pixels.p0_c2 = rgb0.blue;
            currentPixel.pixels.p0_c3 = rgb0.green;

            currentPixel.pixels.p1_c1 = rgb1.red;
            currentPixel.pixels.p1_c2 = rgb1.blue;
            currentPixel.pixels.p1_c3 = rgb1.green;

            stream_in_48.write(currentPixel);
            currentPixel.user = 0; //for rest of Pixels
        }
    }
}
```

Código A.8: En este código tenemos la función stream_to_image, la cual obtiene los coeficientes de color de los pixeles que le llegan por streaming y los escribe en una varible que muestra como imagen a la salida.

```
void stream_to_image(string filename, stream<video_stream>& stream_out_48){
    Bitmap image;
    //vector <vector <Pixel> > bmp; //defined global
    Pixel rgb0, rgb1;
    video_stream currentPixel;
    // Convert from AXI4-Stream to Matrix
    for (int i=0; i < IMAGE_HEIGHT; i++){
        for (int j=0; j < IMAGE_WIDTH; j+=2){
            currentPixel = stream_out_48.read();
            rgb0.red = currentPixel.pixels.p0_c1;
            rgb0.blue = currentPixel.pixels.p0_c2;
            rgb0.green= currentPixel.pixels.p0_c3;

            rgb1.red = currentPixel.pixels.p1_c1;
            rgb1.blue = currentPixel.pixels.p1_c2;
            rgb1.green= currentPixel.pixels.p1_c3;

            bmp[i][j] = rgb0;
            bmp[i][j+1] = rgb1;
        }
    }
    image.fromPixelMatrix(bmp);
    image.save(filename);
}
```

UAM
UNIVERSIDAD AUTONOMA
DE MADRID