TRABAJO FIN DE GRADO

INGENIERÍA EN INFORMÁTICA

Descripciones C/C++ para desarrollo de SoCs en FPGAs

Autor

Raúl Ruiz Bueno

Directores

Begoña del Pino Prieto

Jesús González Peñalver

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, junio de 2017

Descripciones de C/C++ para desarrollo de SoCs en FPGAs

Autor
Raúl Ruiz Bueno

Directores
Begoña del Pino Prieto
Jesús González Peñalver

Descripciones C/C++ para desarrollo de SoCs en FPGAs

Raúl Ruiz Bueno

Palabras clave: FPGA, Co-diseño, Hardware/Software, Procesamiento de Imagen, Video, Tiempo real, Zybo, Zynq, Xilinx, C/C++, SDSoC, Vivado, Standalone, Síntesis de alto nivel.

Resumen

El objetivo de este proyecto, es hacer un estudio sobre las posibilidades que existen en la actualidad para desarrollar sistemas en *FPGAs* (*Field Programable Gate Array*) con arquitecturas *SoCs* (*System on a Chip*) basadas en procesadores ARM, haciendo uso de la herramienta SDSoC desarrollada por Xilinx y basada en un entorno Eclipse. Donde podemos implementar dicho sistema con lenguajes de alto nivel, como C y C++. Además, en la herramienta podemos indicar el código que veamos oportuno para que sea ejecutado por hardware reconfigurable, según nosotros le indiquemos. Este proyecto se desarrollará sobre una plataforma implementada y sintetizada previamente en Vivado. La aplicación carece de sistema operativo(*standalone*), por tanto, en muchos procedimientos no se han podido usar directamente librerías como *OpenCV* o funciones que precisaban de un sistema operativo (SO), como acceder a ficheros. La forma de estudiar cómo mejoran las prestaciones sintetizando en el hardware ciertas funciones frente a que se ejecuten en el software, será mediante filtros de imagen sobre un video entrante en tiempo real. La FPGA captará un flujo de video de entrada que ira almacenando haciendo uso de un VDMA (*Video Direct Memoriy Access*) asignado al HDMI. La resolución del video será de 1920x1080 pixeles. El hardware reconfigurable o el procesador de la FPGA, según seleccionemos, procesará cada fotograma aplicándole el filtro de imagen que elijamos. Cada fotograma se envía a un VDMA de salida donde será emitido por el puerto VGA de la FPGA para poderlo visualizar en un monitor. Para añadirle funcionalidad a la plataforma y hacer un estudio más profundo de todo lo que se puede hacer con esta herramienta (SDSoC) he añadido un codificador de imágenes JPEG, que nos guarda una imagen JPEG en la tarjeta SD cuando pulsemos la opción de hacer una foto. Otra funcionalidad interesante con la que cuenta esta plataforma, es que los filtros de imagen están montados en una arquitectura de *plugins*. En este caso siguiendo una plantilla, cualquier persona que quiera usar la plataforma, puede crear un filtro, recompilar y ese filtro se añadirá automáticamente al sistema, sin que el programador tenga que saber nada de la implementación interna. Así ganamos en flexibilidad y escalabilidad. Al final de este documento leeremos acerca de los resultados obtenidos en este proyecto. Tanto ventajas como desventajas de ejecutar el código en el procesador frente a hacerlo en el hardware reconfigurable, donde podremos ver las altas prestaciones que nos ofrece el hardware reconfigurable. Obteniendo ganancias entre el 27,74 y el 200,72 (dependiendo del filtro de imagen que se aplica) en el hardware en comparación a ejecutarlo en el software. Además de presentar resultados de prestaciones, se analizará la ocupación en la FPGA de la implementación de cada filtro. Terminaremos con una visión de las posibles vías futuras, como implementar el sistema para otra FPGA de la misma familia pero con un mayor número de recursos, implementar el sistema sobre un sistema operativo Linux o usando un sistema operativo Linux reprogramar la FPGA de forma dinámica para poder tener implementados para el hardware reconfigurable un mayor número de *cores IP*.

C/C++ descriptions for SoCs development in FPGAs

Raúl Ruiz Bueno

Keywords: FPGA, Hardware/Software Co-design, Image Processing, Video, Real time, Zybo, Zynq, Xilinx, C, C++, SDSoC, Vivado, High Level Synthesis, Standalone.

Abstract

The target of this project is to do a study of possibilities that we have to develop system on System on Chip (SoC) architectures based on ARM processors, using the tool SDSoC. SDSoC belongs to Xilinx and it is based on Eclipse environment, where we can develop the system with high-level languages, like C and C++. Besides, this tool will pass the code that we want to the reconfigurable hardware, as we indicate. This project will be developed on a platform implemented and synthesized previously in Vivado, exporting the hardware files to be using by SDSoC. This application does not have any operative system (standalone) therefore, in many of cases and functions could not be using directly libraries like OpenCV or system functions, like access to a file. The way to improve performance synthesizing on reconfigurable hardware some functions versus to execute this in the processor with the software, will be using image filters over input video stream. The FPGA will capture a stream of input video that the platform store in one VDMA, assigned to the HDMI port. The size of this video stream can be up to 1920x1080 pixels. The reconfigurable hardware or the FPGA processor, depending of the selection that is choose by physical switch in the board, will process every frame adding an image filter that we choose; some of those filters are grayscale, sepia and sobel edge. Each frame is sent to an output VDMA where will be issued by the FPGA's VGA port to be watched in one display. To add functionality to the platform and do a deeper study of all the things that we can do using this tool (SDSoC) I have added one JPEG encoder, which save a JPEG image in the SD card when we select the take picture option. Other interesting functionality that the platform has, is the plugins architecture where are mounted the image filters. In this case following a template, anyone that wanted to use the platform, can make a filter, rebuild and this filter will be adding automatically to the system, without the programmer having to know anything about the intern implantation. Thus, we obtain more flexibility and scalability. At the end of this document, we will read about the results obtained in this project. As the advantages and disadvantages of execute the code in the processor versus the reconfigurable hardware, where we will see the high performance that offers the reconfigurable hardware, obtaining improvements in performance between 27,74 and 200,72 (depending of the image filter) executing on the reconfigurable hardware compared to running it on the software. In addition to presenting performance results, will analyze the occupation in the FPGA of the hardware implementation of each filter. We will end up with a vision of the posible future ways, how implement the system for another FPGA of the same family, implement this over a Linux operative system or using the operative system Linux to reprogram dynamically the FPGA to be able to have built a large number of IP cores.

Yo, **Raúl Ruiz Bueno**, alumno de la titulación Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 74742190X, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Raúl Ruiz Bueno

Granada a 18 de junio de 2017.

D. **Begoña del Pino Prieto,** Profesora del Departamento de Arquitectura y tecnología de computadores de la Universidad de Granada.

D. **Jesús González Peñalver,** Profesor del Departamento de Arquitectura y tecnología de computadores de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Descripciones C/C++ para desarrollo de SoCs en FPGAs***, ha sido realizado bajo su supervisión por **Raúl Ruiz Bueno,** y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 18 de junio de 2017.

Los directores:

Begoña del Pino Prieto **Jesús González Peñalver**

Agradecimientos

Durante la realización del grado he tenido la suerte de conocer a grandes personas. A las que siempre las recordaré en esta etapa de mi vida.

En primer lugar, tengo que agradecer a mis padres. Durante todo este tiempo he recibido el cariño posible y el apoyo constante de ambos. Además del sacrificio económico que supone estudiar fuera de casa.

A mis dos tutores Begoña y Jesús. Muchas gracias por haberme dirigido el proyecto y por todo el esfuerzo adicional que os ha supuesto, que no ha sido poco. He aprendido muchísimo de vosotros, espero que los que vengan detrás mía sepan apreciar toda la experiencia que tenéis. Me siento muy afortunado de que hayáis sido mis tutores.

A Héctor, mi hermano. A pesar de la gran diferencia de edad que hay entre nosotros, has hecho muy entretenidas y amenas las esperas durante las numerosas compilaciones del sistema.

A mis amigos de toda la vida, muchas gracias por vuestro apoyo, interés y momentos agradables.

A mis amigos de la carrera. Me alegro mucho de haberos conocido, de haber compartido tantas emociones juntos y de haber trabajado a vuestro lado. Os deseo lo mejor y que cada uno de vosotros alcancéis vuestra meta.

En último lugar a Carmen. Has sido la persona que más me ha apoyado. Has sabido calmarme en los momentos de estrés. Te has alegrado conmigo con las buenas noticias. Has estado a mi lado pasara lo que pasara. Muchas gracias de todo corazón.

Gracias de nuevo a todos y cada uno de vosotros.

Índice de contenidos

1. Motivación e introducción.

Los primeros años de este grado son algo abstractos. Muchas asignaturas de matemáticas y física, donde no le ves sentido a ninguna. Otras, que no sabes si volverás a ver algo parecido en tu vida. Todo esto cambió cuando llegué a tercero y elegí especialidad.

Me decanté por la especialidad de ingeniería de computadores por la asignatura de primero “Tecnología y organización de computadores”. Donde me sedujo la forma que tenían componentes electrónicos y físicos, como son las puertas lógicas a realizar operaciones más complejas. Pudiendo ser diseños muy escalables.

Cuando llegué a la especialidad empecé a disfrutar de la carrera. Además, comencé a interesarme por como ciertos componentes electrónicos podían interactuar con el mundo real con ayuda de un programa que les indicara como hacerlo.

En este punto conocí el mundo de las FPGAs, unos dispositivos en los que, si se diseñaban bien ciertos sistemas, podrían tener altas prestaciones. Tengo que decir que el hardware de altas prestaciones, es una de mis debilidades en este ámbito. Ver como algo que has diseñado, capta información del mundo real, la procesa e interacciona con el mundo físico en pocas fracciones de segundo, es de las cosas más satisfactorias que yo creo que le pueden pasar a un ingeniero de hardware.

Me pasé todo el curso previo al proyecto de fin de grado, pensando en hacer algo relacionado con este mundo. Quería ponerle el broche final al grado con algo que realmente me gustara, algo en lo que tengo la intención de dedicarme, al menos, una parte de mi vida laboral como ingeniero de hardware.

Por esta razón, cuando mi tutora me propuso realizar un proyecto de captura y procesamiento de video en tiempo real, no me lo pensé demasiado. Sabía que iba a ser un camino largo y complicado. Pero era el proyecto perfecto para mí, un proyecto para disfrutarlo y demostrar todos los conocimientos adquiridos durante el grado. Un proyecto que cada vez que iba avanzando, más retos iban saliendo a flote, donde solucionándolos, he llegado a grandes niveles de satisfacción personal y profesional.

Hasta ahora el diseño de sistemas embebidos ha sido un campo de trabajo propio de los ingenieros de hardware. La razón, es debido a que hay que tener una serie de conocimientos previos y consideraciones específicas, para poder implementar aplicaciones para dichos sistemas. También es necesario el conocimiento de ciertos lenguajes de programación hardware de bajo nivel, como son el VHDL y el Verilog.

Hace unos años las FPGAs eran dispositivos que no estaban al alcance de todo el mundo, ya sea porque eran caras o difícil de conseguirlas. Pero actualmente, ha habido un importante crecimiento entre la comunidad de desarrolladores para este tipo de dispositivos.

Si navegamos por internet, podemos hacernos con una de estas placas, por precios que empiezan en los 70 euros más o menos. Según nuestras necesidades podemos adquirir FPGAs de mayor tamaño, pero también tendrán un incremento en el precio.
Este incremento de dinero puede ser un gran desperdicio si no tenemos un gran conocimiento en sus lenguajes de programación hardware. Aunque esto puede cambiar.

La intención con la que se eligió una FPGA de Xilinx, integrada en la Zybo Zynq-7000, es porque Xilinx, nos ofrece entre sus herramientas, una que es bastante interesante para ingenieros de software que tengan nociones en C y C++, que se llama SDSoC.

SDSoC se creó con la intención, además apoyado por las facilidades que pueda dar un entorno Eclipse, de que podamos diseñar e implementar sistemas en FPGAs, simplemente escribiendo nuestro código C y C++. Pero lo más interesante de esta herramienta es la opción de pasar cualquier función que nos interese, cumpliendo ciertos requisitos, al hardware reconfigurable, consiguiendo una notable mejora en las prestaciones.

Para la realización de un proyecto de este tipo es necesario usar una plataforma hardware diseñada y exportada de Vivado. Que será donde se ejecutará la aplicación.

La forma de poner en práctica esta herramienta y poder explorar sus posibilidades, ha sido mediante la implementación de una plataforma que simule las funciones de una cámara fotos. Más exactamente, que el flujo de imágenes que va recibiendo por el objetivo sea procesado y devuelto por una pantalla.

Esto puede ser bastante provechoso para ciertos campos que requieren un procesamiento de imágenes en tiempo real para realizar acciones, como puede ser la conducción autónoma o una máquina industrial de clasificación de fruta por color.

Aparte de la parte de potencia computacional, también intervienen aspectos económicos en ámbitos energéticos, ya que evitando la sobrecarga del procesador conseguiremos ahorrar energía, muy provechoso en sistemas que usen baterías. Por otra parte, nuestro sistema puede ganar en tiempo de ejecución, con esto me refiero a que, si un sistema ejecuta cierta parte del código en el hardware, el procesador puede realizar otras tareas necesarias.

A consecuencia de todo lo anterior, este enfoque usando la herramienta SDSoC, conjuntamente con Vivado, hemos conseguido una plataforma de procesamiento de imagen en video.

A lo largo de este documento haremos un recorrido en el ámbito de las FPGAs, comentando sus características y herramientas de trabajo, terminando en una visión completa del sistema que se ha realizado para este proyecto.

En el punto 1 y sus sub-apartados, daré una visión sobre las FPGAs y sus herramientas de diseño.

En los puntos 2 y 3, tenemos los objetivos de este trabajo, los materiales utilizados y la metodología que hemos seguido.

El punto 4 contiene toda la información técnica del sistema implementado, su parte software y hardware.

1.1. SOCs en FPGAs

La mejor forma de abordar este apartado es comenzando por que es un System on Chip (SoC).

Un SoC es un circuito integrado que tiene todos los componentes necesarios mínimos para montar un sistema, como puede ser un *smartphone*, por ejemplo.

Los elementos mínimos que componen un SoC son:

- Microprocesador de uno o más núcleos. En caso de tener más de uno, los conocemos por las siglas MPSoC (*Multiprocessor System on Chip*).
- Bloques de memorias ROM, RAM, EEPROM y FLASH.
- Relojes.
- Interfaces externas como puede ser micro USB, GPIO (*General Purpose Input Output*), Ethernet…
- Convertidores analógicos-digitales y viceversa.
- Reguladores de voltaje.
- Una estructura de bus, donde se realizan las conexiones de los dispositivos.

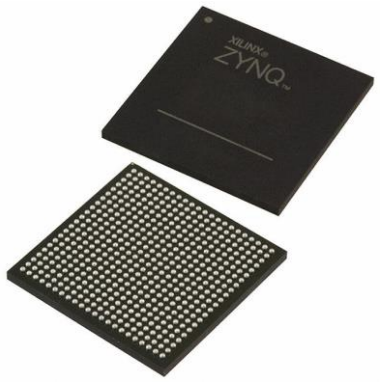


Figura 1, detalle de SoC. Ref [3]

El término de FPGA va a ser referenciado muy a menudo a lo largo de este documento, pero ¿qué es realmente?
De forma sencilla, las FPGAs son unos chips reprogramables. Usando unos bloques lógicos predefinidos y rutas programables, se pueden configurar para implementar alguna funcionalidad hardware.

1.1.1 Arquitectura SoC Xilinx.

Xilinx nos ofrece dos tipos de SoC, el Zynq 7000 y el Zynq Ultrascale.

1.1.1.1 Arquitectura Zynq 7000

En la figura 2, tenemos una vista más clara de lo que exactamente compone este SoC.

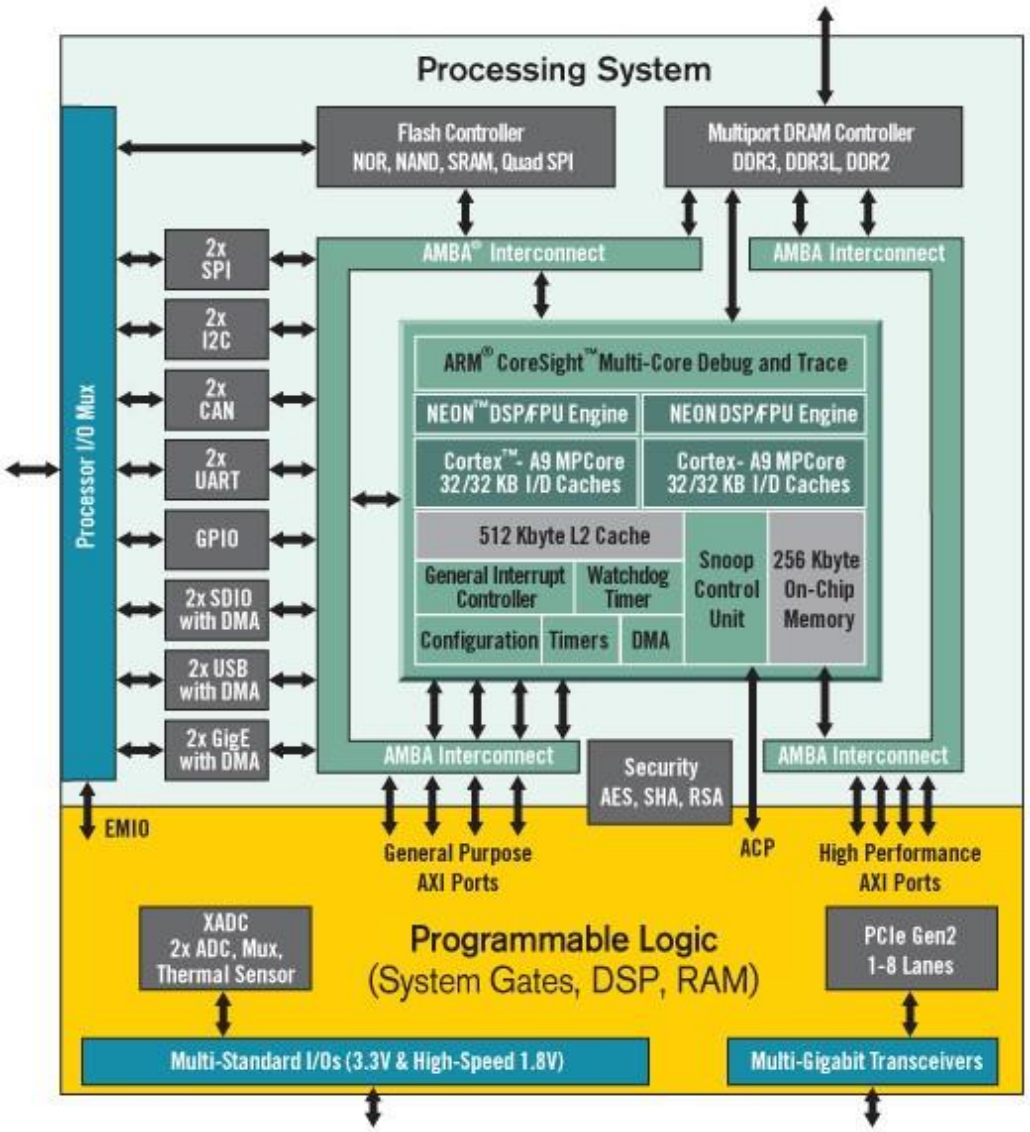


Figura 2, detalle arquitectura Zynq 7000. Ref [16]

Como hemos mencionado en el apartado anterior, contiene todos los elementos mínimos. A destacar, el ARM A9 Cortex de dos núcleos. Todos ellos están conectados y se comunican mediante el bus AMBA.

Lo interesante de esta arquitectura es que también contiene una lógica programable (la zona amarilla) Donde están todas las puertas, DSP y RAM. La comunicación entre el procesador ARM y la FPGA se hace a través del bus AXI.

El bus AXI tiene dos tipos de puerto, uno de propósito general y otro de alto rendimiento.

También podemos ver que este SoC nos ofrece bastantes interfaces de conexión con el exterior, que usan el bus AMBA para comunicarse con el procesador:

- 2x SPI
- 2x I2C
- 2x CAN
- 2x UART
- GPIO
- 2x SDIO con DMA
- 2x USB con DMA
- 2x GigE con DMA

Este SoC es el que se encuentra integrado en la placa de desarrollo Zybo, la utilizada durante la realización del proyecto.

1.1.1.2 Arquitectura Zynq Ultrascale

Como podemos ver en la siguiente imagen, la arquitectura de este SoC es mucho más compleja que el Zynq 7000. Esto se debe a que la arquitectura Ultrascale está orientada a sistemas de tiempo real.

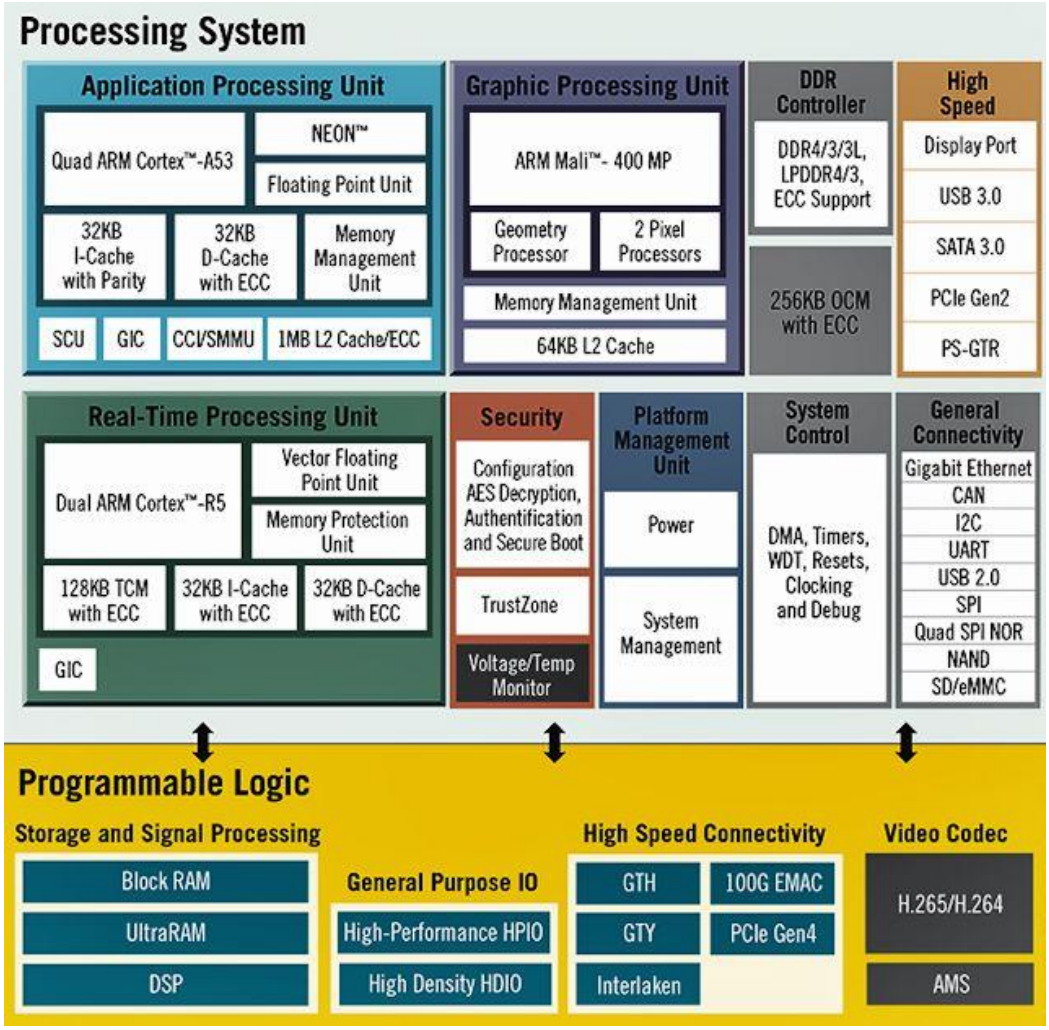


Figura 3, arquitectura Zynq Ultrascale. Ref [4]

A simple vista tiene casi los mismos elementos que la anterior, pero por ejemplo en el apartado de conexiones, contamos con interfaces USB 3.0 y PCIe.

Lo más destacable de esta arquitectura es el *core* IP que tiene para seguridad y el procesador dedicado ARM R5 para procesamiento de tiempo real.

Además de que contamos con una GPU (*Graphics Processor Unit*), que está destinada a la aceleración del código o al procesamiento de imágenes.

En la parte de la FPGA podemos ver un códec de Video de alta definición y conexiones de alto rendimiento.

1.1.2 Arquitectura SoC Intel

Intel tiene cuatro familias de SoC: Stratix 10 SoC, Arria 10 SoC, Arria V SoC y Cyclone V SoC [12]. Aunque ninguna de estas ha sido usada para este proyecto.

1.1.2.1 Intel Stratix 10 SoC

Esta FPGA es la que tiene más prestaciones entre todas las familias de FPGAs SoC.

En la figura 4 podemos ver la arquitectura interna de este SoC:

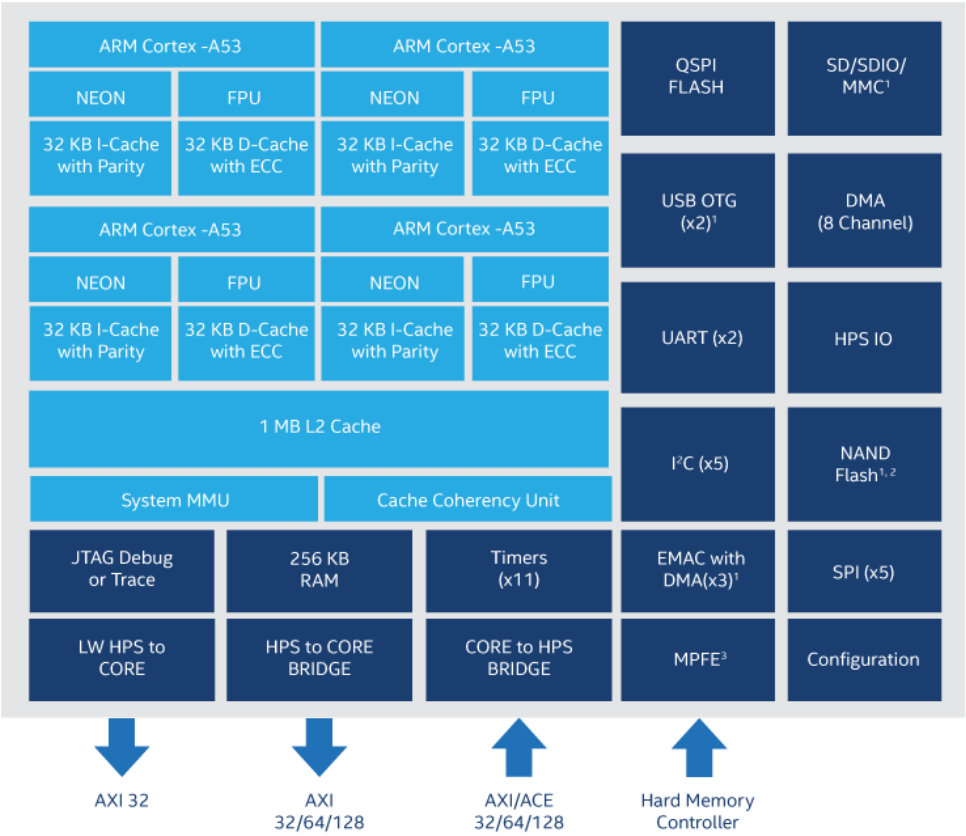


Figura 4, Arquitectura Intel Stratix 10 SoC [13]

De esta arquitectura se puede destacar que tiene un procesador ARM de 4 núcleos

con 2 niveles de caché.

1.1.2.2 Intel Arria 10 SoC

Esta FPGA es la segunda mejor en cuanto a prestaciones. Comparando con la FPGA del punto anterior observamos que en vez de tener 4 núcleos solamente tiene 2. Mantiene los dos niveles de caché, como observamos en la figura 5.

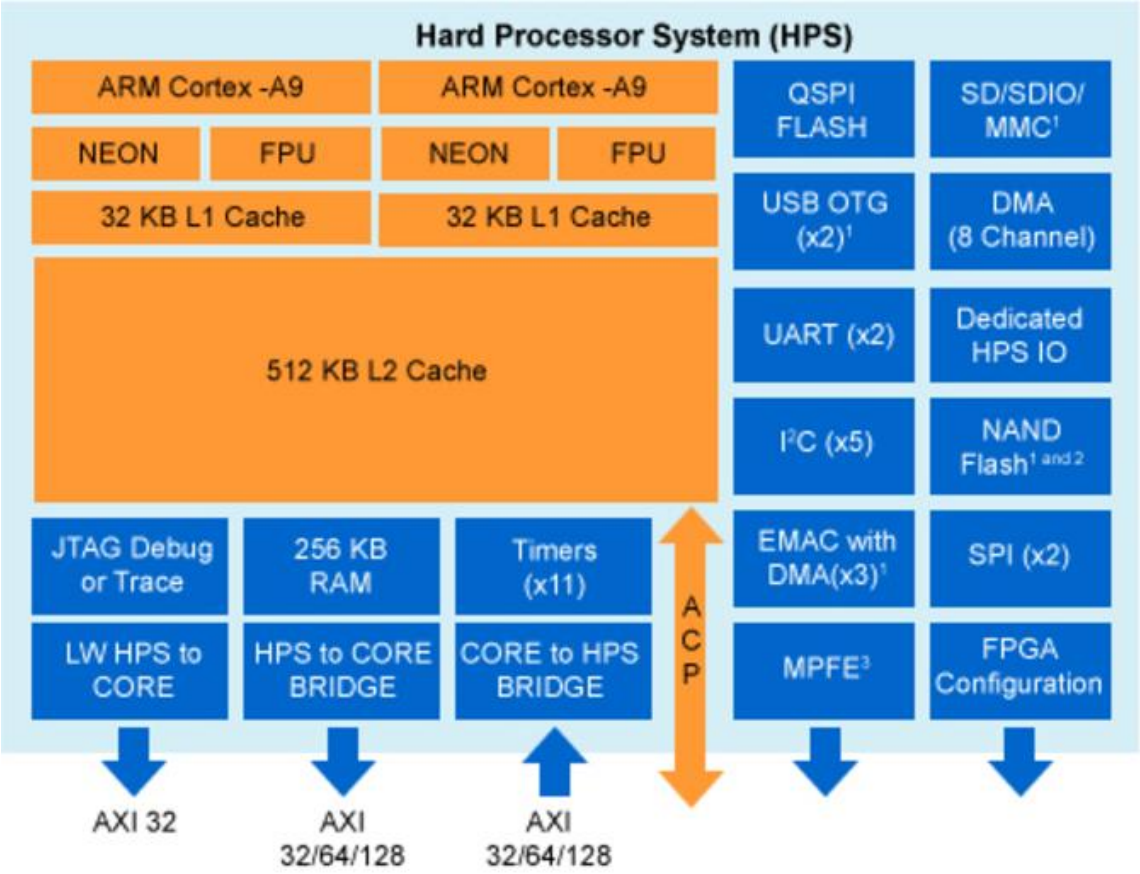


Figura 5, Arquitectura Intel Arria 10 SoC [14]

1.1.2.3 Intel Arria V SoC

Esta FPGA pertenece al escalón más bajo de prestaciones en esta gama de FPGAs. Su arquitectura es como la Intel Arria 10 SoC, pero tiene menos memoria RAM, 64KB frente a 256KB que tiene la Arria 10. Además, la parte de la FPGA está más limitada en cuanto a componentes. [15].

1.1.2.4 Intel Cyclone V SoC

La Cyclone V es la FPGA más pequeña de todas las basadas en SoC. Tiene menos prestaciones que la Intel Arria V.

1.2. Herramientas de desarrollo de SOCs en FPGAs

Hace unos años, el desarrollo de sistemas en FPGA podía ser una tarea bastante tediosa, ya que las interfaces de las herramientas no eran demasiado intuitivas.

Registers Transfer Level (RTL) es un nivel de abstracción donde el circuito es descrito como una colección de registros de almacenamiento, ecuaciones Booleanas, lógicas de control (como if-else) y secuencias de eventos. Esto se describen con lenguajes HDL como VHDL o Verilog.

La síntesis RTL lógica convierte la representación RTL a una mezcla de registros y ecuaciones Booleanas mediante una serie de minimizaciones y optimizaciones. Para finalmente, generar el diseño a nivel de puertas lógicas.

Una de las dificultades implementando sistemas para FPGAs, es la integración de los diferentes módulos. Antes de la aparición de las herramientas de síntesis de alto nivel, la síntesis RTL era mucho más lento y costoso. Ya que un ingeniero de hardware tenía que diseñar el sistema usando lenguajes HDL (*Hardware Description Languajes*) para crear las representaciones del circuito. Los desarrolladores necesitaban que el tiempo de salida al mercado en el mínimo posible, entonces aparecen las herramientas de síntesis de alto nivel.

A lo largo de este apartado, trataré de resaltar todas las opciones que tenemos para trabajar con FPGAs. Haciendo especial hincapié en las herramientas de Xilinx, ya que han sido estas las que he utilizado para la realización de este proyecto. Terminando con una breve introducción con alguna herramienta de Intel / Altera.

1.2.1 Herramientas de Xilinx

Xilinx es la compañía que ha desarrollado el SoC que está integrado en la Zybo. Este proyecto ha sido diseñando usando la versión 2016.2, que era la más actual en la fecha que empecé a trabajar. Esta marca nos ofrece las siguientes herramientas:

1.2.1.1 Vivado

Vivado [9] es una de las herramientas más importantes, completas e interesantes que tiene Xilinx para el desarrollo en FPGAs. Su principal función dentro del desarrollo del hardware es que, mediante una interfaz bastante intuitiva, podemos diseñar, implementar y sintetizar plataformas hardware mediante la combinación de ciertos componentes.

Los componentes que se integran en estas plataformas se conocen como “IP *cores*” (*Intellectual Property core*). Cada IP *core*, es un bloque que contienen una implementación hardware de alguna operación específica, descrita en lenguajes HDL. La propia herramienta nos proporciona un gran catálogo que podemos usar para nuestros diseños.
La integración de estos IP *cores* nos dará como resultado una plataforma hardware completa.

Entre las funciones de esta herramienta encontramos las siguientes:

- Definición de la plataforma hardware.
- Simulación de la plataforma hardware.
- Análisis RTL, donde podemos el esquema de puertas lógicas del sistema diseñado.
- Síntesis de nuestro sistema. Donde implementamos el código VHDL de los componentes al esquema de puertas lógicas.
- Programación y depuración en la placa.
- Emisión de informes, por ejemplo, consumo de recursos y prestaciones.

En este proyecto esta herramienta se ha usado para actualizar la plataforma hardware desde la versión 2015.4 a la 2016.4 (Revisar apéndice A), ya que Xilinx ya nos ofrecía dicha plataforma funcional [1].

1.2.1.2 Vivado HLS

Cuando en el apartado anterior hemos comentado que Vivado integra *cores* IP para crear plataformas hardware, se ha mencionado la existencia de un catálogo con de *cores* IP. ¿Pero qué pasa si no está ahí el que necesito?

Vivado HLS [10] es la solución a esa pregunta. Esta herramienta sirve para implementar *cores* IP a medida, usando lenguajes de alto nivel como C/C++.

Esta herramienta tiene una interfaz propia de entornos eclipse. Aparte de poder programar en C/C++ podemos añadir directivas hardware a nuestro código para optimizar el diseño y ganar en prestaciones.

Esta herramienta ha sido esencial para el desarrollo de este proyecto. La razón es porque SDSoC, que se introducirá en el siguiente apartado, llama a Vivado HLS cuando necesita pasar al hardware reconfigurable alguna función, ya que esta herramienta es la que se encarga de implementar y hacer la síntesis de alto nivel a los *cores* IP.

1.2.1.3 SDSoC

Terminadas las introducciones a las herramientas para implementar y diseñar la plataforma hardware, llegamos a la parte de la implementación software de la aplicación.

Cuando comencé este proyecto, SDSoC [7] estaba recién publicada. Estamos hablando que, a fecha de este proyecto en este momento, esta herramienta tiene solamente un año.

Xilinx lanzó SDSoC como alternativa las demás herramientas, para que los ingenieros que no tuvieran experiencia implementando sistemas hardware en FPGAs pudieran desarrollar, en lenguajes de alto nivel como C/C++, y acelerar aplicaciones software. Así finalmente serian implementadas para hardware reconfigurable.

Esta herramienta además ofrece la opción de poder utilizar una plataforma definida en Vivado o una genérica. También podemos elegir entre hacer aplicaciones que

funcionen sobre un sistema operativo Linux o hacerlas sin sistema operativo (standalone). Esta segunda opción es la que usa este proyecto.

La interfaz que tiene SDSoC es también heredada de entornos eclipse, de forma que es bastante intuitiva de usar y nos da muchas opciones de personalización para nuestro proyecto.

Realmente lo que hace esta herramienta es que el código que nosotros pasamos al hardware, lo manda en segundo plano al Vivado HLS de forma que SDSoC crea los scripts tcl que usa Vivado HLS para crear *cores* IP y automáticamente integra dicho *core* IP al diseño de la plataforma sobre la que estamos trabajando.

Aparte de poder implementar código en SDSoC tenemos un *profiller* integrado, donde podemos ver el gasto de recursos que hace cada sección del código. También contamos con una serie de informes que contienen la ocupación de la plataforma en la FPGA y el rendimiento obtenido hardware frente a software.

En la figura 6, podemos ver un diagrama de referencia del entorno de desarrollo de SDSoC.

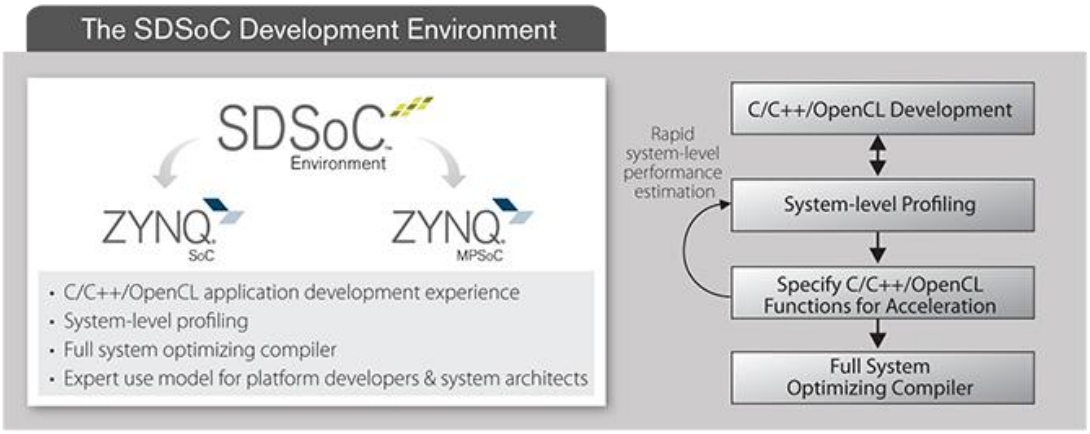


Figura 6: Entorno de desarrollo SDSoC [7]

En el apartado 4.4 profundizaremos más en el uso de esta herramienta.

1.2.1.4 SDAccel

Antes de comenzar con este punto, hay que saber que OpenCL es un lenguaje de programación basado en C. Es multiplataforma. La diferencia esencial con C, es que está optimizado para el paralelismo.

OpenCL se creó con la intención acelerar cálculos en arrays y matrices en GPUs. En Altera (ahora es de Intel), por su parte, se pensó que podía ser una buena funcionalidad que podían tener su FPGAs, y así conseguir más prestaciones en el cálculo matricial.

SDAccel es la solución que ofrece Xilinx para poder implementar códigos en OpenCL en algunas familias de sus FPGAs. Por esta razón la necesidad de esta herramienta nace de la tendencia actual que hay por usar grandes clústeres compuestos por FPGAs para tareas cómputo intensivo en vez de servidores convencionales.

Las ventajas principales que nos ofrecen usar este tipo de arquitecturas son las siguientes:

- Consumo energético bajo.
- Baja latencia.

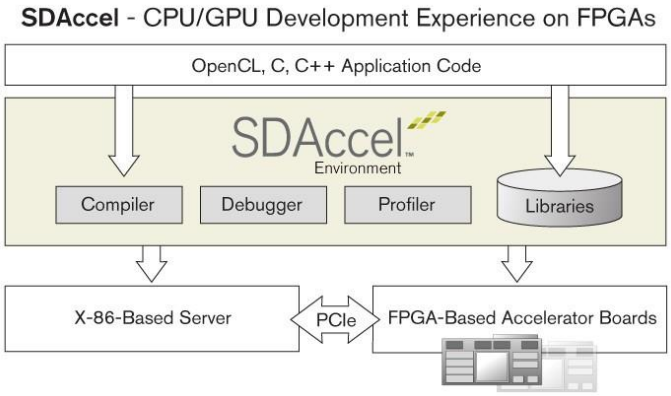


Figura 7, Flujo de desarrollo SDAccel. Ref [5]

El caso es que SDAccel no solamente está pensado para implementar sistemas en servidores. También está muy orientada para ámbitos como tratamiento de imagen en tiempo real, ya que las GPUs están muy optimizadas para hacer procesamientos de datos en matrices, consiguiendo unas prestaciones bastante altas.

Este proyecto no contiene nada que trabaje con OpenCL, por tanto, no ha sido necesario su uso.

1.2.2 Intel

Por otra parte, Intel es la competencia directa con Xilinx. Nos ofrece otra serie de herramientas para trabajar con sus familias de FPGAs.
Como el proyecto es para una tarjeta de Xilinx, este conjunto de herramientas no las he usado. Por tanto, haré una breve mención de cada una de ellas

Intel Quartus Prime Design es la herramienta que se encarga de crear una plataforma hardware para las FPGAs de Intel. A diferencia de Xilinx, esta interfaz es mucho más compleja de usar. En Vivado podíamos montar el sistema simplemente arrastrando los cores IP a la ventana de diseño de la plataforma, pero aquí hay que escribirla en lenguaje HDL.

Para trabajar de manera más productiva sobre los SoC basados en ARM, Intel ofrece La herramienta Intel SoC FPGA Embedded Development Suite. Es sólo para aplicaciones software.

Usa un entorno eclipse como interfaz y la implementación de los programas para estos SoCs se hacen en lenguajes de alto nivel como C/C++. Las aplicaciones que ejecuta

estos SoCs, pueden ser tanto sobre Linux como standalone.

Desde esta herramienta tenemos tanto la opción de depurar el código como una pequeña terminal donde podemos ir comprobando la salida que nos da la ejecución de la aplicación.

Esta división de FPGAs antes tenía el nombre de Altera, hasta que Intel la compró. Altera fue la pionera en usar OpenCL en sus FPGAs. Intel por su parte ha seguido lanzando herramientas para el desarrollo de esta tecnología.

Intel FPGA SDK sirve para diseñar *cores* IP, para sus FPGAs, que contengan alguna implementación de OpenCL.

1.3. Descripciones C/C++ para síntesis de componentes hardware.

La síntesis de alto nivel es una técnica que mediante lenguajes de alto nivel como C/C++, se realiza una implementación hardware de los módulos descritos en dicho código. Vivado HLS [11] es la herramienta que se encarga de esta síntesis.

Como pequeña introducción en el flujo de desarrollo de la herramienta SDSoC, una vez tenemos nuestro código escrito en C/C++, en ciertos módulos como puede ser un doble bucle for, podemos escribir unas directivas de Vivado HLS, seleccionamos esa función para que se implemente para el hardware reconfigurable, compilamos y SDSoC llama a Vivado HLS para sintetizar los componentes e integrarlos en un sistema completo.

Vivado HLS está compuesta por una serie de bibliotecas diseñadas de forma que sea fácil hacer síntesis RTL a partir de funciones modeladas en C/C++.

Estas librerías son las siguientes:

- Arbitrary Precision Data Types Library
- HLS Stream Library
- HLS Math Library
- HLS Video Library
- HLS IP Library
- HLS Linear Algebra Library
- HLS DSP Library

La forma de añadir estas librerías en este tipo de proyectos, se hace incluyendo el “include” en las cabeceras.
La librería “Arbitrary Precision Data Types Library” es el mecanismo que usa el HLS para definir la anchura de los datos, optimizando el tamaño. En C los tipos de datos van en potencias de 2, entonces si necesitamos un multiplicador de 17 bits, Vivado HLS optimiza el sistema de forma que no tengamos que incluir uno de 32bits. Su principal ventaja es conseguir tamaños de datos más pequeños.

“HLS Stream Library” se encarga de las transferencias de datos cuando hay un flujo continuo de datos. Esto lo hace proporcionando estructuras de datos para el almacenamiento, lectura y escritura.

Por otra parte “HLS Math Library” da soporte para la síntesis de las bibliotecas math.h y cmath.h. Incluye funciones de punto flotante.

Para el procesamiento de video tenemos la librería “HLS Video Library”. Contiene un gran número de funciones útiles para el procesamiento de imagen y video. Los tipos de datos de esta biblioteca son fácilmente sintetizables para ser integrados en *cores* IP.

La implementación de los *cores* IP son directamente inferidos desde el código C mediante la librería “HLS IP library”. Esta librería elige los canales, anchura y arquitectura de los *cores* IP que van a ser integrados.

“HLS Linear Algebra Library” está orientada para el uso de operaciones con matrices en dos dimensiones. El nivel de optimización para la implementación RTL depende como este escrito el código C y las directivas que se usan.

Las funciones de construcción de bloques para el modelo del sistema de DSPs en C++ se encuentran en la biblioteca HLS DSP Library.

1.4. Estructura de la memoria

Este documento está dividido en 9 apartados:

En la primera parte se introduce el proyecto y se menciona la motivación que me ha llevado a realizarlo. Así como una pequeña introducción de algunos ejemplos de arquitecturas SoC y a las herramientas de diseño utilizadas para trabajar con ellos.

La segunda parte de esta memoria presenta la metodología seguida y los materiales usados para la realización de esta memoria. También está dentro de este punto, la planificación temporal y el presupuesto de este proyecto.

En tercer lugar, se analiza el diseño realizado de la plataforma. Tanto la parte hardware, como la parte software.

La cuarta y última parte de esta memoria abarca las conclusiones personales obtenidas tras la realización de este proyecto y vías futuras de desarrollo. Esta parte termina con la bibliografía utilizada.

2. Objetivos del trabajo.

El objetivo de este proyecto de fin de grado es explorar y estudiar las opciones que nos ofrece la herramienta SDSoC de desarrollo en SoC de Xilinx para FPGAs basadas en procesadores ARM.

Se pretende analizar las prestaciones que podemos obtener implementando un sistema en SDSoC. Además, se comprobará el grado de dificultad en el uso desde el punto de vista de un ingeniero de software, que no esté especializado en FPGAs, implementando una aplicación en C/C++ sobre una plataforma hardware.

Las secciones que requieran de cómputo intensivo serán compiladas para el hardware reconfigurable de la FPGA y así conseguir mayores prestaciones.

Para el desarrollo de este proyecto se usará una placa de desarrollo de Digilent con un SoC de Xilinx: Zybo Zynq-7000.

La Zybo es una placa de desarrollo bastante pequeña y de coste reducido. Pero tiene una gran potencia de cálculo y una FPGA integrada.

Finalmente, partiendo desde una plataforma hardware de captura de video creada por Digilent [1], los objetivos marcados para la realización de este proyecto fin de grado han sido:

- Implementación de algunos filtros de imagen para su procesado. Con la intención de comparar las prestaciones al ejecutar estos módulos en el hardware reconfigurable o el procesador. Siendo este objetivo el más prioritario.

- Creación de una arquitectura de plugin, de forma que podamos añadir y eliminar filtros de imagen a demanda. Este objetivo muestra que con esta herramienta se pueden hacer aplicaciones flexibles y escalables.

- Implementación de una funcionalidad para la captura de imágenes, de forma que, llamando a una función, la imagen que ha sido procesada la guarde en la tarjeta SD de la FPGA con una compresión JPEG.

La implementación de los filtros ha sido terminada por completo, teniendo disponibles 5 filtros de imagen diferentes:

- Escala de grises.
- Detección de bordes.
- Filtro sepia.
- Énfasis de color.
- Filtro Laplaciano.

La implementación de dichos filtros está en una arquitectura de *plugins* basada en herencia. Estando este objetivo completado en su totalidad. Con esta funcionalidad añadida, si algún ingeniero quiere usar esta plataforma, solo tendría que segur la plantilla de diseño del *plugin*, añadirla en el directorio correspondiente y recompilar.
Demostrando la flexibilidad y escalabilidad del sistema.

La intención del último objetivo ha sido darle más funcionalidad, que como los anteriores está completado. El codificador JPEG convierte las imágenes RGB que han sido procesadas por algún filtro, ya sea en el hardware reconfigurable o en el software,

en un fichero .JPEG en la tarjeta SD de la FPGA. Teniendo como opción poder ver la imagen filtrada en otro dispositivo.

Estos objetivos los estudiaremos en más profundidad en el apartado 6.3, donde se explicará el desarrollo de la aplicación software.

3. Materiales y metodología.

3.1 Materiales

3.1.1 Ordenador portátil.

Para la programación hardware/software, pruebas de funcionamiento, pruebas de rendimiento y captura de la fuente de video por HDMI he utilizado el equipo MSI GE60 2OE. Las prestaciones de este equipo son las siguientes:

- Intel core I7-4700MQ a 2.40 GHz.
- 16 GB de memoria RAM DDR3.
- Tarjeta gráfica Nvidia GTX 765m.
- Sistema operativo Windows 10.
- Salida de vídeo HDMI.

3.1.2 Placa de desarrollo Zybo Zynq – 7000.

Esta placa es comercializada bajo la compañía Digilent [6], pero el SoC está firmado por Xilinx. Dentro de la familia de FPGAs de la serie 7000, esta es exactamente la ZYNQ XC7Z010-1CLG400C.

La FPGA de desarrollo Zynq 7010 AP SoC tiene las siguientes características:

- Un procesador ARM Cortex A9 de doble núcleo a 650MHz.
- 512 MB de RAM DDR3 con 8 canales DMA.
- Memoria FLASH de 128 MB.
- 28000 celdas lógicas.
- Bloque de memoria RAM de 240KB.
- 80 DSPs (*Digital Signal Processor*).
- 60 BRAM (*Block RAM*).
- 17600 LUTs (*Lookup Table*).
- 35200 FFs (*Flip-Flops*).

Además, los puertos periféricos de los que dispone esta placa de desarrollo son los siguientes:

- JTAG por micro USB, para comunicación por puerto serie y programación.
- Puerto USB 2.0.
- USB OTG.
- Ethernet 1GB.
- Control de periféricos I2C.
- Puerto HDMI I/O.
- Puerto VGA salida.
- Jacks para entrada y salida de audio.
- 4x Switches.
- 6x Botones.

- 5x Leds.
- Puerto micro SD

3.1.3 Monitor

Monitor Asus VK24, con puerto VGA. Acepta resoluciones hasta 1920x1080 píxeles.

3.1.4 Herramientas de desarrollo

Para el desarrollo de la aplicación se han utilizado las herramientas Vivado, Vivado HLS y SDSoC.

3.2 Metodología

Se ha seguido una metodología basada en prototipos para la realización este proyecto. La razón de haberlo hecho de este modo es porque es el que más se ajustaba a un proyecto incremental de este tipo. Contiene los siguientes apartados:

1. Acercamiento, apartado 3.2.1
2. Plataforma base, apartado 3.2.2
3. Generar nuevos filtros, apartado 3.2.3
4. Creación de un codificador para JPEG, apartado 3.2.4
5. Integración de arquitectura de *plugins*, apartado 3.2.5
6. Documentación.

En este proyecto podemos distinguir tres bloques de desarrollo. El punto 1, que recoge todo el entrenamiento previo en las herramientas. El punto 2, 3, 4 y 5 que son los puntos de desarrollo de la plataforma. El punto 6, donde se ha documentado este proyecto.

3.2.1 Acercamiento.

En el acercamiento lo primero que se ha hecho ha sido instalar las herramientas necesarias como el SDSoC [7] y la suite de Vivado [9].
El siguiente paso ha sido la familiarización con la herramienta SDSoC, leyendo sus manuales y realizando sus tutoriales con la placa Zybo.

3.2.2 Elección del punto de partida.

Una vez tenía conocimiento y algo de destreza sobre las herramientas que iban a ser usadas para la realización de este proyecto, pasamos a ver varios ejemplos de plataformas que nos ofrecía Xilinx.

El que fue descartado, era una aplicación de tratamiento de video YUV, que se ejecutaba sobre un Linux embebido. Se encontraba entre los ejemplos en la creación de un proyecto para la Zybo en SDSoC.

Por otra parte, elegimos como punto de partida una plataforma de tratamiento de video [1]. Lo interesante de este proyecto, es que, a diferencia del otro, este capta las imágenes por el cable HDMI y lo devuelve por el VGA. Por tanto, este podía llegar a ser un proyecto más interesante.

En esta parte se tuvo que actualizar la plataforma elegida a una versión más actual, esta parte termina con la finalización de prototipo número 1.

3.2.3 Generar nuevos filtros.

La plataforma inicial contaba con dos filtros iniciales, uno de transformar la imagen de entrada a una en escala de grises y otro de detección de bordes.

Para darle más opciones de funcionalidad a la plataforma, se crearon tres filtros más. Que son: filtro sepia, filtro Laplaciano y énfasis de color. Añadiéndolos a la misma función junto a los otros que teníamos anteriormente, consiguiendo el prototipo número 2.

3.2.4 Creación de un codificador JPEG.

Habiendo cumplido el objetivo de añadir algunos filtros más, lo interesante era poder guardar las imágenes en un fichero resultado dentro de la tarjeta SD. El factor más desafiante de este apartado ha sido que, al carecer de un sistema operativo, el codificador ha tenido que ser *standalone*, con funciones de muy bajo nivel, sin poder usar bibliotecas implementadas para este tipo de procesos, como OpenCV. El hito conseguido en esta parte es el prototipo número 3.

3.2.5 Integración de la arquitectura de plugins.

Dado que teníamos varios filtros en una misma función, lo interesante era separarlos, para conseguir una aplicación más modular. Pero donde, además, se pudieran cargar más filtros diferentes a demanda.

Finalmente, esta arquitectura se ha implementado, basada en herencia. Porque con la FPGA tan limitada y al ser una aplicación *standalone* era la mejor opción.

Terminando así con el sistema final, que es el prototipo número 4.

4. Sistema completamente configurable para procesamiento de imagen

Aprovechando las ventajas de diseño e implementación que nos puede dar una FPGA, opté por trabajar en un sistema de procesamiento de imagen en tiempo real.

La plataforma hardware sobre la que funciona dicho sistema ha sido diseñada en Vivado. Cuenta con un *core* IP de entrada que capta la imagen del puerto HDMI de algún dispositivo periférico y lo guarda en un VDMA. Este VDMA le envía los datos al procesador que hará los procesamientos sobre la imagen que nosotros le indiquemos.

El procesador coloca dichos datos en otro VDMA, que está conectado a un *core* IP, el cual transmite las imágenes por el puerto VGA, pudiéndolo conectar a cualquier monitor para ver los resultados.

La aplicación software por otra parte, se encarga de inicializar todos los accesos a los diferentes dispositivos que hay definidos en el diseño de Vivado. Una vez terminada esta inicialización, daríamos paso a la ejecución de nuestra aplicación de procesamiento de video. Donde podemos elegir la resolución de salida, los tipos de filtro de imagen y una funcionalidad para tomar una imagen y guardarla en formato JPEG en la tarjeta SD de la placa.

4.1 Planificación y presupuesto

Este proyecto ha tenido una duración de un año completo, comenzando en Julio de 2016, con una media de aproximadamente 5 horas a la semana.

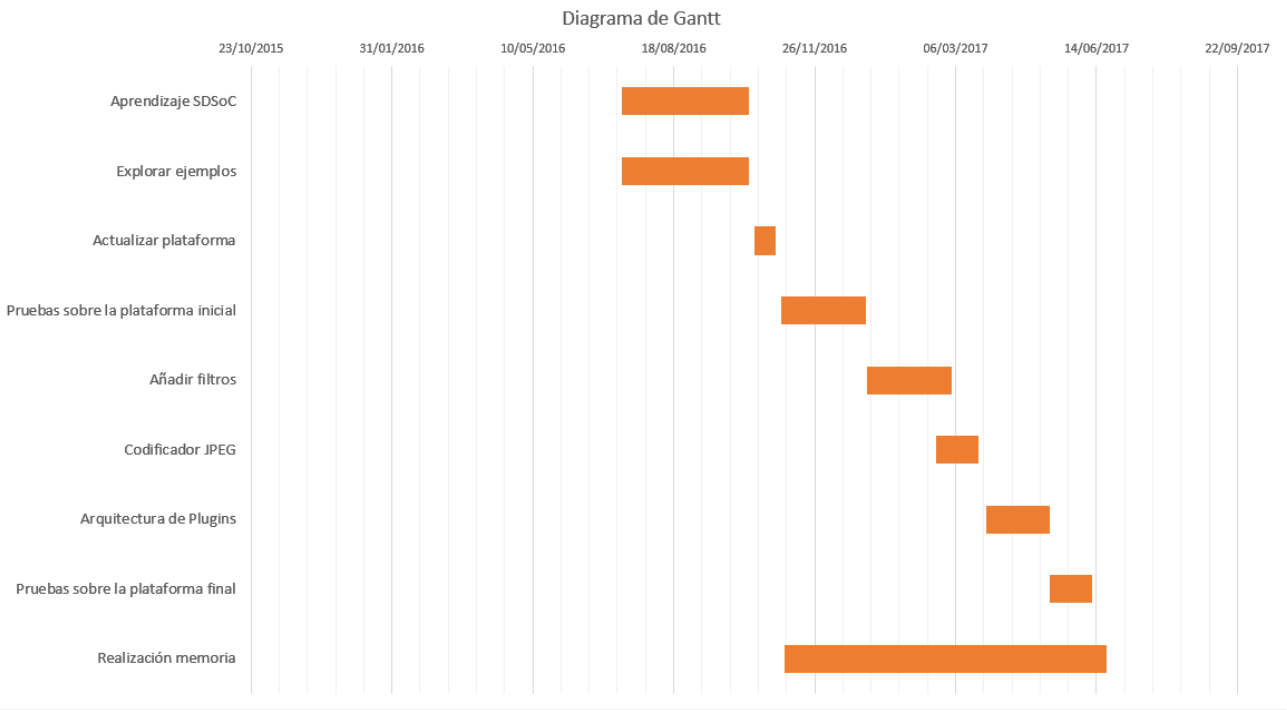


Tabla 1: Diagrama de Gantt

En referencia a la Tabla 1, ciertas tareas como añadir filtros e implementar el codificador JPEG se podrían haber hecho en paralelo. Pero al ser un equipo unipersonal era preferible ir cumpliendo objetivos antes de pasar a la siguiente tarea.

El desglose de la planificación durante este periodo de tiempo quedaría de la siguiente forma:

- Julio, agosto y septiembre: Aprender a manejar SDSoC y explorar diferentes plataformas que nos ofrecía Xilinx como pequeños ejemplos por dónde empezar a trabajar.

- Octubre, noviembre y diciembre: Una vez elegido el camino a seguir y teniendo más conocimiento en las herramientas de desarrollo de Xilinx, el siguiente paso era actualizar la plataforma de video a la versión de las herramientas de desarrollo que he usado, ya que esta plataforma se encontraba en unas versiones más bajas. Una vez actualizado todo y teniendo los objetivos bien definidos, comencé a trabajar con la aplicación software en algunas tareas. Por ejemplo, haciendo pequeñas pruebas para comprobar la ganancia que recibíamos de la plataforma si ciertas partes del código las sintetizábamos en el hardware de la FPGA en vez de ejecutarlo en el procesador.

- Enero, febrero, marzo, abril, mayo y Junio: Una vez había aprendido a manejar bien las herramientas de Xilinx y dominaba la plataforma era el momento de mejorar ciertas partes del código y añadir funcionalidades. Dichas funcionalidades han sido nuevos filtros de imagen, un codificador JPEG para guardar imágenes en la tarjeta SD de la FPGA y una arquitectura de *plugins* para darle flexibilidad a la plataforma, de tal manera que cualquier persona que tenga acceso a la plantilla del *plugin*, pueda diseñar uno, colocarlo en el directorio correspondiente y recompilar la plataforma sin necesidad de que tenga que saber cómo funciona todo lo demás.

El siguiente punto a tratar es el presupuesto de este proyecto, el cual tendría el siguiente desglose:

- FPGA: Zybo Zynq-7000 ARM/FPGA SoC board * 220,99€
- Cable HDMI * 3,95€
- Cable VGA * 5,75€
- Mano de obra: 39€/hora x 175 horas x 21% IVA 8.258,25€
- Licencia HL System Edition* 3.853,73€

- Total 12.342,67€

Los elementos marcados con * llevan el IVA incluido de sus distribuidores

4.2 Plataforma hardware

Llegados a este punto es el momento de analizar cómo funciona la plataforma hardware sobre la que se ejecuta nuestra aplicación [1].

La parte correspondiente al diseño y desarrollo de esta plataforma en Vivado es

ofrecida por Xilinx como ejemplo de aplicación SDSoC. Mi función en esta parte ha sido actualizarla a la versión correspondiente, ya que inicialmente pertenecía a la versión 2015.4 y yo he usado la versión 2016.2, la más reciente cuando empecé el proyecto.

4.2.1 Arquitectura y diseño de la plataforma

Como introducción a este apartado, para que el diseño de la plataforma sea más fácil de entender, voy a explicar de una forma sencilla como es el flujo de los datos, haciendo uso de la figura 8:

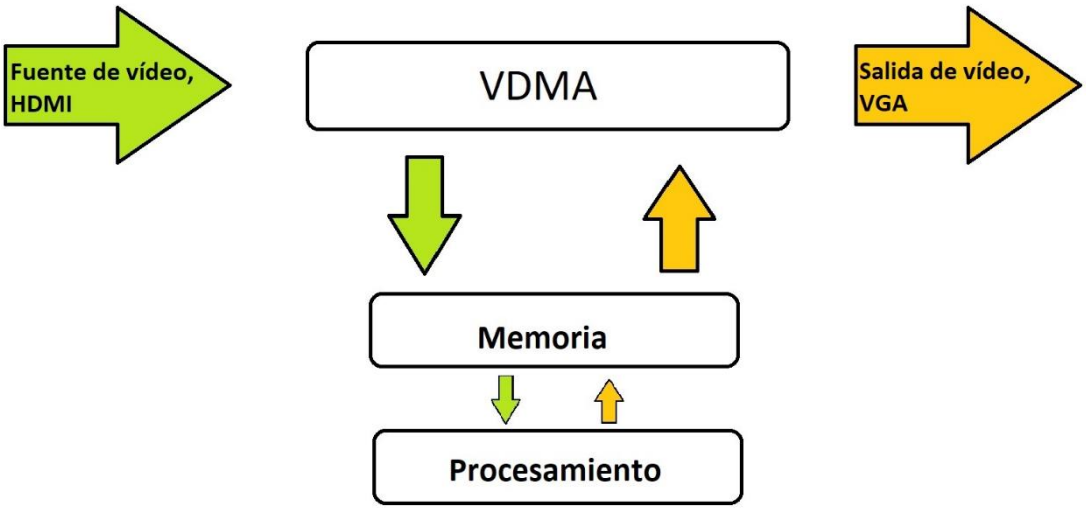


Figura 8, flujo de los datos de imagen

La plataforma recibe un flujo de entrada constante de video, a través del puerto HDMI, que mediante el bus AXI, lo almacena en memoria donde el VDMA gestiona la direcciones para que el procesador o el hardware reconfigurable puedan usar los datos. Una vez acabado dicho procesamiento, los datos se vuelven a colocar en la memoria y a través del bus AXI los envía por puerto VGA.

Una vez tenemos clara esta pequeña introducción podemos avanzar un paso más en el estudio de la arquitectura de la plataforma. En la figura 9 podemos ver detalladamente el diseño de la arquitectura de esta plataforma. Comprobando cómo se conectan los diferentes dispositivos que intervienen en el flujo de los datos. Lo más interesante de este diseño es el bloque “Sobel Engine”, que es la función que nosotros hemos elegido en la herramienta SDSoC para que sea implementada para el hardware reconfigurable.

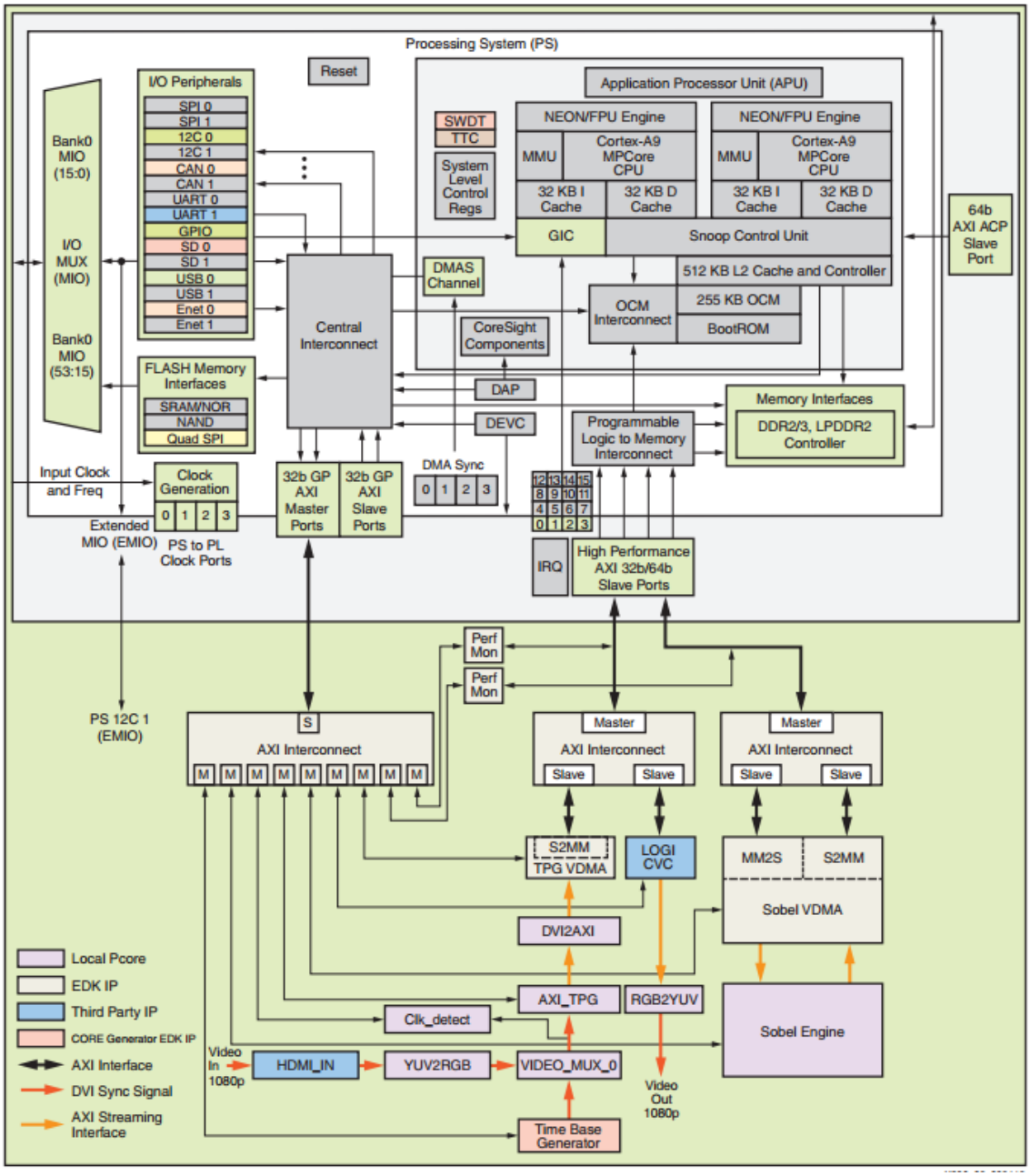


Figura 9: Plataforma video vivado [16]

En la herramienta de Vivado podemos ver el diseño de una forma simplificada:

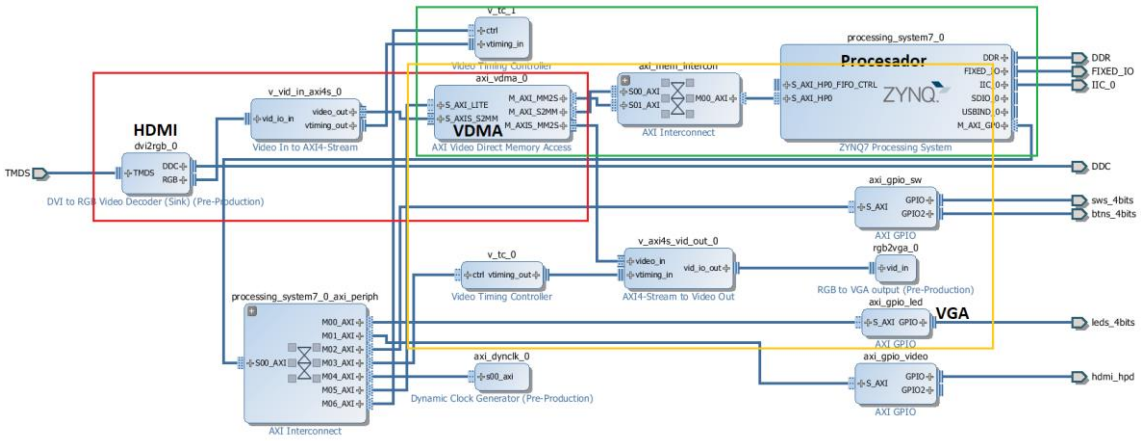


Figura 10, plataforma de video simplificada.

Ahora que tenemos una primera aproximación del diseño real de la plataforma, vamos a ver el camino que sigue el flujo de datos.

Como hemos visto antes, el primer paso es llevar los datos captados por el puerto HDMI, que corresponde al elemento dvi2rgb_0, al VDMA [17], axi_vdma_0, haciendo uso del bus AXI [20], en la figura 7 pertenece al área marcado en rojo, la podemos ver con más exactitud en la figura 11:

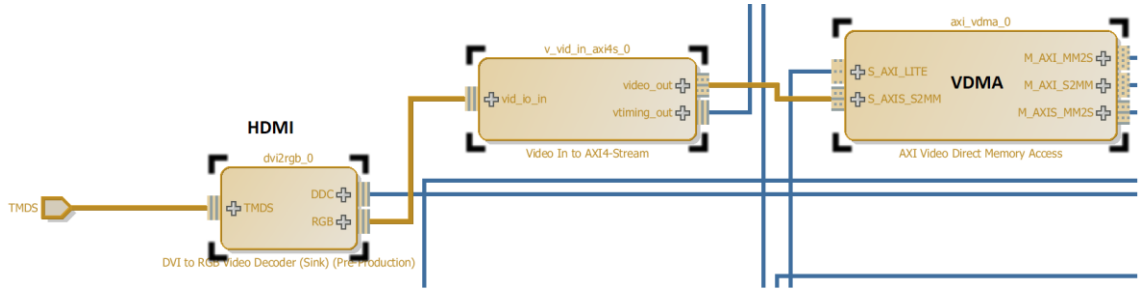


Figura 11, detalle captura por HDMI.

Desde el VDMA se depositan los datos en la memoria, como podemos ver en la figura 12, que serán recogidos posteriormente por el procesador. En esta parte tenemos dos opciones, implementarlo para el procesador o para el hardware reconfigurable. Si elegimos la opción de hacerlo en el hardware reconfigurable, más tarde en SDSoC, al compilar la plataforma SDSoC creará los scripts TCL de los *core IP* que contendrán la implementación del proceso que hayamos elegido para que los pueda usar Vivado HLS y así implementarlos. Una vez hecho este proceso, devolvemos el resultado de los datos usando el VDMA y siguiendo el camino de forma inversa.

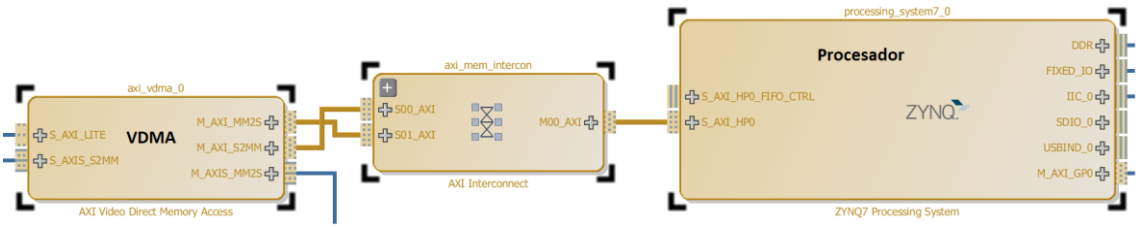


Figura 12, detalle envío de datos al procesador.

Finalmente, el VDMA gestionará el envío de los datos al puerto VGA, elemento rgb2vga_0, detallado gráficamente en la figura 13.

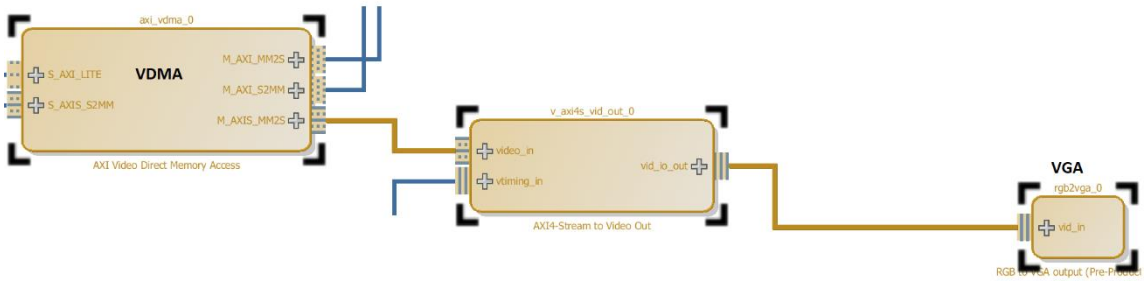


Figura 43, salida de datos por VGA

Cerrando así el estudio del diseño de la arquitectura para la plataforma.

4.3 Aplicación software

Como he adelantado en la introducción y en los objetivos, la finalidad de esta aplicación software es procesar un flujo de video entrante, procesarlo y emitirlo para que lo podamos ver en una pantalla. Además, podemos elegir si queremos que ciertas partes del código se ejecuten en el procesador o en el hardware reconfigurable.

Esta aplicación consta de tres partes bien diferenciadas:

1. Procesamiento de imagen (filtros).
2. Codificador de imágenes JPEG.
3. Arquitectura de *plugins*.

En el diagrama de casos de uso, figura 14, podemos ver la relación entre un usuario y las funcionalidades del sistema.

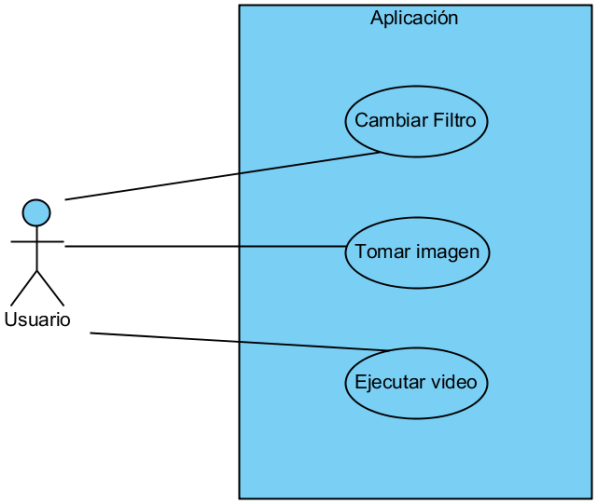


Figura 14: Diagrama de casos de uso.

4.3.1 Procesamiento de imagen.

Este es el punto más importante del desarrollo del proyecto. La razón es porque el análisis de prestaciones lo haremos en esta parte, que realmente es donde se encuentra el procesamiento más costoso.

En el ciclo de vida de este proceso, podemos distinguir tres partes fundamentales, captura de la imagen, procesamiento de la imagen y emisión de la imagen.

En el ejemplo de Xilinx sobre el que se está desarrollando este proyecto [1] la parte de captura y emisión ya estaba implementada. Además, contiene dos filtros que son escala de grises y detección de bordes, que han sido completados con tres filtros más.

Partiendo de esta situación inicial vamos a ver cómo funciona esta parte.

Como hemos visto en el apartado 4.2, donde se explicaba cómo funciona la plataforma hardware, tenemos un *core IP* encargado de tomar las imágenes que nos van llegando por el puerto HDMI físico de la placa.

Este *core IP* guarda los datos en un *array* de C del tipo *unsigned int* de 16 bits. La idea de usar este tipo de dato es porque realmente lo que va captando el *core IP* son valores RGB y de que solo necesitamos valores positivos.

En el *array* los datos se almacenan de forma que se intenta optimizar al máximo el espacio de la placa. Con esto quiero decir que, el tamaño de inicialización del *array* es de 1920x1080. Lo lógico sería que fuera 1920x1080 x3 (cada componente RGB), pero al usar un tipo de dato de 16 bits sin signo, en la misma celda se guardan las 3 componentes RGB, ahorrando mucho espacio.

Una vez tenemos almacenado todos los datos en el *array* se lo pasamos a la función de procesamiento de la imagen.

Esta función, aparte del puntero al *array*, recibe un parámetro que nos permitirá seleccionar el filtro que queramos entre los que tengamos registrados en la arquitectura de *plugins*. Llamando al filtro que hemos seleccionado.

En este punto hemos llegado a la parte donde se hace el procesamiento del *array*. Esta función va recorriendo todo el *array* realizando las operaciones necesarias sobre cada pixel RGB, punto que será tratado con más detalle en el apartado 4.4. Lo más interesante de esta parte, es que podremos elegir si ejecutar la función de procesamiento en el procesador o en el hardware reconfigurable, consiguiendo unas prestaciones bastante notables en este último. En el apartado 5 trataremos los resultados dichas ejecuciones.

Terminando con la emisión del *array* por puerto VGA mediante el *core IP* encargado de esta tarea.

4.3.2 Codificador de imágenes JPEG.

De manera que pudiéramos seguir estudiando las capacidades de esta herramienta se ha realizado un codificador de imágenes JPEG para poder guardar las imágenes procesadas en la tarjeta SD. Al trabajar en una plataforma *standalone*, no se ha podido

usar ninguna biblioteca de terceros como *OpenCV*, que habría resuelto este apartado fácilmente. Xilinx para el acceso a ficheros en *standalone* no usa la biblioteca *fstream*, usa una biblioteca propia, *xilffs* [20]. Este codificador se ha hecho basándome en el de la siguiente referencia, que sigue la estructura de cualquier codificador de imágenes JPEG [19].

4.3.3 Arquitectura de plugins.

Con la intención de hacer un sistema flexible y escalable se ha diseñado una estructura de *plugins* para los filtros. De tal forma que cualquiera que sepa escribir un filtro en C/C++ y siguiendo una plantilla de diseño solo tenga que implementar un fichero de cabecera y un fichero fuente, guardarlos en el directorio donde se alojan el resto de filtros y recompilar. Sin que se tenga que preocupar de cómo está implementada el sistema.

La arquitectura elegida para esta estructura ha sido la herencia. La clase *Plugin* es una clase abstracta que implementa métodos estáticos, así cuando sea llamada existirá el mismo objeto para todos.

El constructor autorregistra al objeto que lo llama en un *array* de punteros tipo *Plugin*. La intención de hacerlo así es porque en las clases hijas, cada vez que se ejecuta el constructor de la clase llama a la clase padre. De esta forma conseguimos que cada filtro se guarde solo en el vector.

Para que esto surta efecto, en el mismo fichero fuente del filtro tenemos que declarar un objeto de esa clase, así se crea y se autorregistra. Para que poner un ejemplo práctico de la implementación de un filtro, en la figura 15 podemos ver la plantilla a seguir para crear el fichero de cabecera de un nuevo filtro y en la figura 16 el fichero fuente:

```
8  #ifndef SRC_ADDONS_FILTER1_H_
9  #define SRC_ADDONS_FILTER1_H_
10
11 #include "../plugin.h"
12
13
14 class Filtro1 : public Plugin {
15     public:
16         Filtro1();
17
18         //Call to HW or SW function
19     void apply (u16 srcFrame[DEMO_PIXELS], u16 destFrame[DEMO_PIXELS], int hwsw);
20 };
21
22
23
24
25 #endif /* SRC_ADDONS_FILTER1_H_ */
26
```

Figura 14: Plantilla fichero de cabecera de un filtro.

```
#include "filter1.h"



#define ABS(x)          ((x>0)? x : -x)
#define length(x) (sizeof(x)/sizeof(x[0]))

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

#pragma SDS data access_pattern(srcFrame:SEQUENTIAL, destFrame:SEQUENTIAL)
void HardwareSobelFilter1(u16 srcFrame[DEMO_PIXELS], u16 destFrame[DEMO_PIXELS]){

}

void SoftwareSobelFilter1(u16 srcFrame[DEMO_PIXELS], u16 destFrame[DEMO_PIXELS]){

}

    Filtro1::Filtro1(){
            name = "nombreFiltro";
            desc = "descripcionFiltro";
        }

   void Filtro1::apply (u16 srcFrame[DEMO_PIXELS], u16 destFrame[DEMO_PIXELS], int hwsw) {
        if(hwsw == 1)
            HardwareSobelFilter1(srcFramen,destFrame);
        else
            SoftwareSobelFilter1(srcFramen,destFrame);
    };

//to create the object decoment this, and this is automatically added to the plugin class
//static Filtro1 f;
```

Figura 15: Pantilla fichero fuente de un filtro.

Una vez montado toda esta arquitectura, la forma de usar esta clase se realiza mediante llamadas a la clase *Plugin*, que contiene funciones para devolver el número de elementos, devolver puntero a objeto filtro a partir de nombre o la posición en el vector de filtros.

Por ejemplo, en la siguiente figura, podemos ver el caso de uso de elegir otro filtro de los que tengamos disponibles en la estructura de plugins.

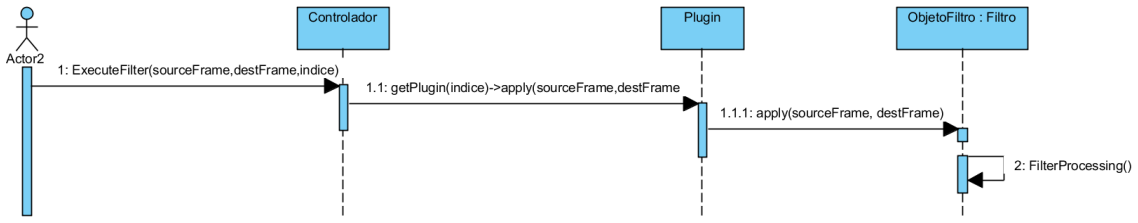


Figura 16: Flujo de la llamada otro filtro.

Y el diagrama de clases de los componentes y funciones implicadas:

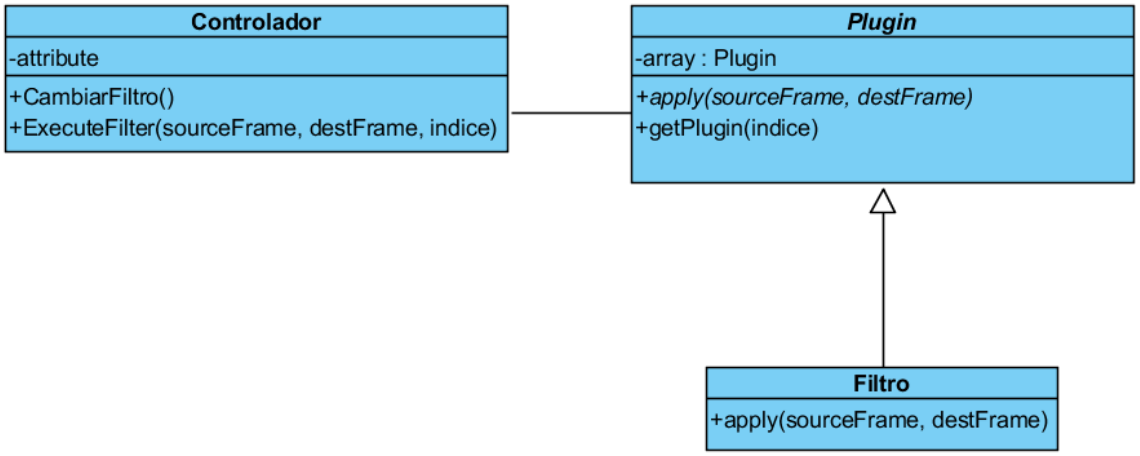


Figura 18: Diagrama de clases usadas en la operación ejecutar filtro.

A diferencia del proyecto inicial [1] que implementaba todos los *plugins* en la misma función. Aquí cada *plugin* se implementa en una función aparte. Esto tiene, como hemos mencionado, ventajas de escalabilidad y de rendimiento. Por otra parte, tiene una gran desventaja, y es que depende mucho sobre la FPGA donde se implementa la plataforma. En la Zybo que es una FPGA bastante limitada, si sintetizamos 2 filtros para el hardware reconfigurable, agota todos los *slices* y no podemos crear el sistema.

4.4 Módulos hardware/software

En el apartado 1.2.1.3 se ha dado una breve introducción acerca de SDSoC, en esta parte se analizará en más profundidad la funcionalidad de SDSoC para implementar funciones para el hardware reconfigurable.

Pues bien, una de las peculiaridades que tiene SDSoC es que algún desarrollador que esté usando esta herramienta puede elegir si implementar funciones (cumpliendo unos requisitos) para el hardware reconfigurable de la FPGA, donde se pueden obtener mejores prestaciones como trataremos en el apartado 5, o ejecutarlas en el procesador.

Elegir que se implemente en el hardware no implica que siempre vayamos a obtener los mejores resultados. De hecho, podemos conseguir peores o iguales resultados que en la ejecución de la misma función en el procesador.

Para conseguir estas mejoras o cumplir ciertos requisitos para la implementación en el hardware hay que añadir ciertos *pragmas* y directivas al código para que Vivado HLS lo interprete correctamente [21].

Para explicar esta parte de una forma más intuitiva, hacemos uso de la Figura 19 que contiene la implementación de la función hardware que calcula la escala de grises de una imagen.

```
#include "filter2.h"

#define ABS(x)          ((x>0)? x : -x)
#define length(x) (sizeof(x)/sizeof(x[0]))

#pragma SDS data mem_attribute(srcFrame:PHYSICAL_CONTIGUOUS|NON_CACHEABLE,
    destFrame:PHYSICAL_CONTIGUOUS|NON_CACHEABLE)
#pragma SDS data access_pattern(srcFrame:SEQUENTIAL, destFrame:SEQUENTIAL)
void HardwareSobelGray(u16 srcFrame[DEMO_PIXELS], u16 destFrame[DEMO_PIXELS])
{
    int xcoi, ycoi;
    u32 frameWidth;

    ap_linebuffer<u8, 3, DEMO_WIDTH> buff_A;
    ap_window<u8,3,3> buff_C;



    for(ycoi = 0; ycoi < DEMO_HEIGHT+1; ycoi++){
        for(xcoi = 0; xcoi < DEMO_WIDTH+1; xcoi++){
#pragma AP PIPELINE II = 1
            u8 temp, grayIn = 0,sepiaIn=0,grayInOld=0;

            u16 rIn, gIn, bIn,rOut,gOut,bOut;
            u16 pxlIn, pxlInOld = 0,pxlOut;
            u16 edge;


            //Metemos en xcoi al linebuffer y guardamos en temp el valor
            //que hemos metido al final del linebuffer
            if(xcoi < DEMO_WIDTH){
                buff_A.shift_up(xcoi);
                temp = buff_A.getval(0,xcoi);
            }
            if((xcoi < DEMO_WIDTH) & (ycoi < DEMO_HEIGHT)){
                pxlInOld = pxlIn;

                //cogemos el pixel de esta posición
                pxlIn = srcFrame[ycoi*DEMO_WIDTH+xcoi];
```

Figura 19: Filtro escala de grises.

El compilador sdscc, que es el que usan los proyectos SDSoC, selecciona el protocolo de control de funciones hardware basado en la estructura del programa, el prototipo de la función hardware y los tipos de argumentos que tiene. Según los datos de entrada que tenga esta función y como está implementada la plataforma hardware, se deben añadir unas directivas u otras.

En la Figura 19, podemos ver que antes de la función hay 2 *pragmas*. El primer *pragma* (SDS data mem_attribute) le indica al compilador que los datos están almacenados en memoria física contigua.

El otro *pragma* sirve para guiar a la interface en la generación para el acelerador. Aquí podríamos recibir dos tipos de parámetros, RAMDOM o SECUENTIAL. Si el argumento especificado (srcFrame y destFrame en nuestro caso) estuviera alojado en la RAM sería RAMDOM. Como estos parámetros pertenecen al *streaming* de video entonces usamos SECUENTIAL, que es el argumento usado para las interfaces de *streaming*.

Ya dentro de los dos bucles for nos encontramos con un *pragma*, el de pipeline. Este

pragma realiza un desenrollado del bucle y paraleliza cada iteración consiguiendo unas prestaciones bastante notables.

```
47
48                  //Descomponemos el punto en sus 3 componentes RGB
49                  rIn = ((pxlIn & 0xF800) >> (11-3)); //5 bits para el rojo
50                  bIn = ((pxlIn & 0x07C0) >> (6-3));  //5 bits para el azul
51                  gIn = ((pxlIn & 0x003F) << 2);      //6 bits para el verde+
52
53                  //Calculamos el valor gris a partir del punto original
54                  grayIn = (rIn * 76+ gIn *150 + bIn*29 +128 ) >> 8;
55                  grayInOld = grayIn;
56
57
58                  if(grayIn > 255)
59                      grayIn=255;
60                  if(grayIn < 0)
61                      grayIn=0;
62
63
64              if(ycoi > 0 && xcoi > 0){
65
66                      destFrame[(ycoi-1)*DEMO_WIDTH+(xcoi-1)] = (((grayInOld) & 0x00F8) << (11-3)) |
67                              (((grayInOld) & 0x00F8) << (6-3)) | (((grayInOld) & 0x00F8) >> 2);
68
69              }
70
71
72              }///IF END
73          }//For2 END
74      }//For1 END
75  }
```

Figura 20: Código fuente escala de grises.

En cuanto al procesamiento de la imagen, figura 20, como hemos mencionado anteriormente a lo largo de este documento, la variable pxlIn contiene el pixel de entrada y está almacenada en un unsigned int de 16 bits. La forma de acceder a cada componente RGB del pixel se hace mediante desplazamiento.

Una vez tenemos el valor de cada componente, calculamos el valor de gris para ese punto y normalizamos. Finalmente guardamos en el *array* el resultado de esta operación recomponiendo el pixel [26] [27] [28].

El filtro sepia y énfasis de color siguen la misma metodología que el filtro de escala de grises.

Para la conversión de la imagen a una tonalidad sepia es necesario pasar primero la imagen a escala de grises. El siguiente paso es convertir cada componente RGB al valor deseado y devolver los nuevos valores calculados en la salida.

```
                rOut = (grayIn+grayIn *112) >> 8;
                gOut = (grayIn+grayIn *66) >> 8;
                bOut = (grayIn+grayIn *20) >> 8;

                //saturar todo
                if(rOut> 255)
                    rOut = 255;
                if(rOut < 0)
                    rOut = 0;
                if(gOut > 255)
                    gOut = 255;
                if(gOut < 0)
                    gOut = 0;
                if(bOut > 255)
                    bOut = 255;
                if(bOut < 0)
                    bOut = 0;



            }
            if(ycoi > 0 && xcoi > 0){

                pxlOut = (((rOut + 4) & 0x00F8) << (11-3)) | (((bOut + 4) & 0x00F8) << (6-3))
                | (((gOut + 4) & 0x00F8) >> 2); //el valor +4 y el  0x00F8 es para normalizar
                destFrame[(ycoi-1)*DEMO_WIDTH+(xcoi-1)] = pxlOut;
            }
```

Figura 21: Procesamiento filtro Sepia.

El filtro de énfasis de color por otra parte en la salida tenemos que comprobar el valor RGB del color que queremos que se acentúe. Para este proyecto he elegido el color rojo con valor *Red* por encima de 100, *Green* por debajo de 100 y *Blue* por debajo de 100. En la siguiente figura podemos ver como se queda la implementación del código a la salida.

```
if(ycoi > 0 && xcoi > 0){
    if(rIn>100 && gIn<100 && bIn <100)
        destFrame[(ycoi-1)*DEMO_WIDTH+(xcoi-1)] = pxlIn;
    else{
        destFrame[(ycoi-1)*DEMO_WIDTH+(xcoi-1)] = (((grayInOld) & 0x00F8) << (11-3))
        | (((grayInOld) & 0x00F8) << (6-3)) | (((grayInOld) & 0x00F8) >> 2);
    }
}
```

Figura 22: Procesamiento filtro énfasis de color.

Los filtros de convolución Laplaciana y detección de bordes requieren del cálculo de cada pixel en escala de grises previamente. Una vez hecho esto, se procesa la imagen haciendo uso de sus respectivas matrices de convolución, ya que para estos filtros el código es el mismo, solo varían dichas matrices. La siguiente figura pertenece al filtro de detección de bordes.

```
/*
 * Procesamiento para la detección de bordes
 */
    //Si no se dan las condiciones
    if( ycoi <= 1 || xcoi <= 0 || ycoi == DEMO_HEIGHT || xcoi == DEMO_WIDTH)
        edge=0;
    else{
        short x_weight = 0, y_weight = 0;
        u8 i, j;

        //Kernel para deteccion de bordes con sobel edge
        const short x_op[3][3] = { {-1,0,1},{-2,0,2},{-1,0,1}};
        const short y_op[3][3] = { {1,2,1}, {0,0,0}, {-1,-2,-1}};

        for(i=0; i < 3; i++){
            for(j = 0; j < 3; j++){

                x_weight = x_weight + (buff_C.getval(i,j) * x_op[i][j]);
                y_weight = y_weight + (buff_C.getval(i,j) * y_op[i][j]);

            }
        }
        edge = ABS(x_weight) + ABS(y_weight);

        if(edge > 200)   edge = 255;
        else if(edge < 100)      edge = 0;

    }
```

Figura 23: Procesamiento detección de bordes.

Llegados a este punto seleccionamos la función deseada para que Vivado HLS la sintetice para el hardware reconfigurable. Para ello dentro del entorno de SDSoC, en la parte del proyecto, hacemos clic derecho sobre la función elegida y seleccionamos *Toggle HW/SW*, como podemos ver en la figura 21. Y compilamos.

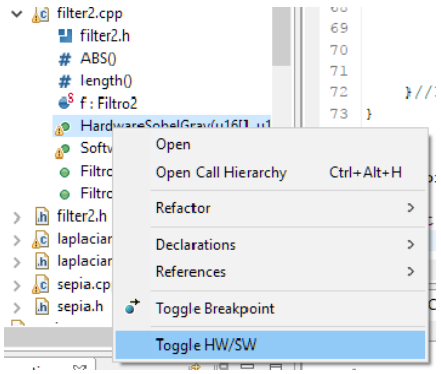


Figura 24: Función hardware.

Con ayuda del *profiller* que viene integrado en esta herramienta podemos ver si la ganancia obtenida pasando ciertas funciones al hardware es mayor que ejecutarlo en el procesador. Por lo tanto, es una funcionalidad bastante útil para decidir que funciones irán implementadas en hardware en la plataforma final.

En el siguiente apartado, capítulo 5, veremos realmente las prestaciones del sistema haciendo uso de estas técnicas de desarrollo.

5. Resultados

En este apartado analizaremos la diferencia de prestaciones implementando las funciones de procesamiento para el hardware reconfigurable y comparándolas con su ejecución en el procesador. Examinaremos la ocupación de la función implementada en el hardware y el sistema completo.

Haciendo uso de la funcionalidad del codificador JPEG en la Figura 25 podemos observar las imágenes resultado para cada filtro de imagen, cada una de ellas con una resolución de 1920x1080.

Figura 25: A original, B escala de grises, C énfasis de color, D filtro sepia, E filtro Laplaciano, F filtro de detección de bordes.

Para hacer las tomas de tiempo de ejecución de los filtros se ha usado la función XTime_GetTime() definido en el fichero de Xilinx xtime_l.c. Siendo usadas del mismo modo que en cualquier otra plataforma:

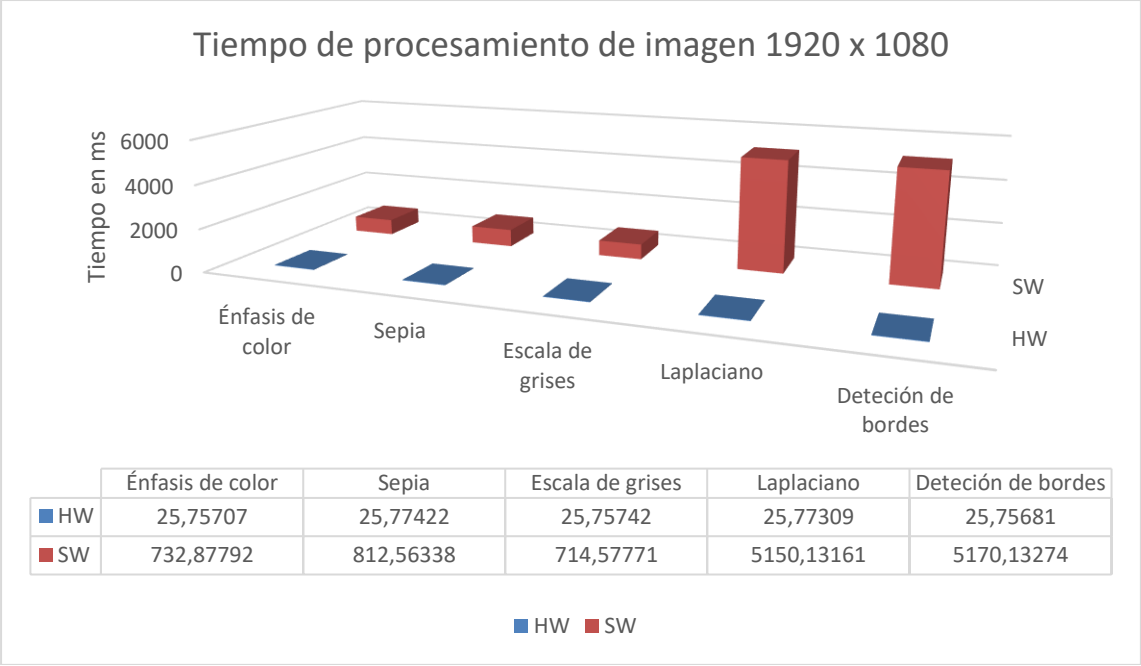
```
XTime_GetTime(&tStartHW);

executeFilter();

XTime_GetTime(&tEndHW);

printf("Output took %.2f us.\n\r",1.0 *(tEndHW - tStartHW) / COUNTS_PER_SECOND/1000000));
```

En el siguiente gráfico podemos observar la gran diferencia existente entre la ejecución de cualquier filtro de imagen en el procesador ARM frente a hacerlo en el hardware reconfigurable.



	Énfasis de color	Sepia	Escala de grises	Laplaciano	Deteción de bordes
HW	25,75707	25,77422	25,75742	25,77309	25,75681
SW	732,87792	812,56338	714,57771	5150,13161	5170,13274

Figura 26: Comparación hardware/software.

El cálculo de la mejora en prestaciones la tenemos en la Tabla 1, donde podemos ver claramente la mejora que recibe el algoritmo de detección de bordes consiguiendo unas prestaciones de 200,72 veces mejor y unos FPS (*Frames Per Second*) de 38,82.

La razón de que los filtros Laplaciano y detección de bordes consigan unas prestaciones tan buenas, es porque al ejecutar en procesador estamos realizando una estructura con 4 *for* anidados, como hemos visto anteriormente en el apartado 4.4. Mientras que, cuando se configura en el hardware reconfigurable, Vivado HLS sintetiza una arquitectura específica que paraleliza los cálculos.

	Énfasis de color	*Sepia*	*Escala de grises*	*Laplaciano*	*Detección de bordes*
Speedup	28,45	31,52	27,74	199,82	200,72
FPS SW	1,36	1,23	1,39	0,19	0,19
FPS HW	38,82	38,79	38,82	38,80	38,82

Tabla 2: Comparación de prestaciones HW/SW

Usar una arquitectura de *plugins* para montar y ejecutar cada filtro, es beneficioso para el sistema. Ya que esto supone una mejora en velocidad de procesamiento con respecto a la implementación en una única función con los cinco filtros a seleccionar.

La desventaja principal de la arquitectura de *plugins* en este sistema es la ocupación en la FPGA. Debido a los pocos recursos que tiene esta FPGA no se ha logrado implementar más de un *core IP* para el hardware reconfigurable a la vez usando la plataforma de *plugins*, a diferencia de tener todos los filtros en el mismo *core IP* que si se ha podido. La razón es que se sobrepasaba el número de *slices* en la FPGA.

Un *slice* es un bloque de componentes configurable en una FPGA de Xilinx. Cada *slice* contiene un número concreto de *LUTs*, *flip-flops* y registros de desplazamiento. La distribución del número de componentes ha sido mencionada en el apartado 3.1.

En las siguientes tablas podemos observar la ocupación que obtenemos de los informes de compilación del sistema. En la Tabla 2 podemos ver el porcentaje de ocupación de la implementación en el hardware de cada función. La Tabla 3 contiene la información de la tabla anterior más los recursos consumidos por la plataforma.

Recurso	Énfasis	Sepia	Grises	Laplaciano	Bordes
DSP	3%	7%	3%	3%	3%
BRAM	0%	0%	0%	2%	2%
LUT	1%	1%	1%	2%	2%
FF	0,60%	0,35%	0,30%	1%	0,89%

Tabla 3: Consumo de recursos de cada filtro.

Recurso	Énfasis	Sepia	Grises	Laplaciano	Bordes
DSP	3,75%	7,50%	3,75%	3,75%	3,75%
BRAM	35,83%	33,33%	33,33%	35,83%	35,83%
LUT	60,18%	63,31%	63,27%	64,23%	64,21%
FF	55,35%	54,60%	54,63%	55,35%	55,15%
Slices	98,16%	99,80%	97,20%	98,16%	98,20%

Tabla 4: Consumo de recursos de cada filtro + plataforma.

6. Conclusiones y vías futuras

En este proyecto hemos hecho un estudio de las posibilidades que nos ofrecen las herramientas de desarrollo de sistemas en arquitecturas SoC basadas en procesadores ARM empotrados en FPGAs. Se ha llevado a cabo mediante el desarrollo de una aplicación de cómputo intensivo para el procesamiento de imágenes en tiempo real.

Cabe destacar que estas herramientas de desarrollo son bastante útiles. No solamente por los resultados obtenidos, sino por el tiempo de desarrollo. Ya que, no hay que olvidar que el desarrollo de este proyecto se basa en una metodología de prototipos. De manera que realizando varias versiones y analizando con el *profiller* se ha llegado a una solución adecuada.

Todos los objetivos propuestos han sido realizados con éxito:

- Creación de nuevos filtros de imagen. Entendiendo el funcionamiento de la plataforma se han implementado tres filtros más.
- Codificador de imágenes JPEG. Aun no teniendo un sistema operativo para que gestione los servicios de acceso a ficheros, se ha conseguido comprimir una imagen de 1920x1080 pixeles en formato JPEG y guardarlo en la tarjeta SD de la FPGA.
- Plataforma de *plugins*. Se ha conseguido montar una arquitectura de *plugins*, sin ayuda de un sistema operativo, de forma que los nuevos filtros se autorregistren y los tengamos disponibles en el sistema.

En cuanto a la arquitectura de *plugins* se ha demostrado que ha sido un punto muy positivo para este proyecto. No solo por la capacidad de flexibilidad y escalabilidad que nos da tener este tipo de soportes, también por el resultado de las prestaciones conseguidas, como por ejemplo en el filtro de detección de bordes, donde se ha conseguido una mejora del 200,72 veces superior implementando la función para el hardware reconfigurable frente a ejecutarlo en el procesador. Esto se debe a que los componentes utilizados en la FPGA para realizar esta operación, es mucho menor que los componentes implicados al tener todos los filtros a seleccionar en el mismo *core*, que implica un mayor tiempo de procesamiento.

SDSoC, tras leer el manual y haber hecho los tutoriales de Xilinx, es una herramienta bastante útil, con una interfaz muy intuitiva. Si lo que se desea hacer no es un sistema sencillo, sino por ejemplo algún tipo de algoritmo de multiplicación de matrices o procesar algún archivo de la tarjeta SD, simplemente se escribe el código como lo haríamos en el PC, darle al botón de lo que queremos que se sintetice para el hardware y compilar. Una vez compilado el sistema, se usa el *profiller* para comparar las prestaciones en la ejecución en el hardware reconfigurable o en el procesador. De esta forma se elige si es más acertado que ciertas funciones se ejecuten en uno u otro. La opción de poder sintetizar funciones para el hardware acorta mucho el tiempo de desarrollo del sistema. La ventaja de usar SDSoC para este tipo de tareas, es que no hay que implementar nada a bajo nivel.

Antes de SDSoC estuviese en el mercado, para implementar sistemas para SoCs con descripciones C/C++ había que diseñar la plataforma con Vivado, sintetizar los módulos hardware con Vivado HLS e integrar la aplicación software con Vivado SDK. Por otra parte, para implementar sistemas con SDSoC es necesario diseñar la plataforma hardware con Vivado, pero desde el mismo entorno se escribe el código de la aplicación. Sigue siendo necesario escribir los módulos hardware con descripciones

que Vivado HLS pueda interpretar para realizar implementaciones hardware eficientes. Pero al usar SDSoC tenemos la ventaja de poder utilizar el *profiller*.

El único punto negativo al que me he enfrentado en este proyecto ha sido que la FPGA usada era bastante limitada y no podía tener sintetizado para el hardware 2 filtros en funciones diferentes.

Migrar un sistema de este tipo a otra FPGA implica rediseñar la plataforma en Vivado para la otra FPGA e implementar la parte de las inicializaciones de los DMAs en el código de SDSoC, ya que las posiciones de la memoria de una FPGA a otra pueden variar.

Como conclusión sobre esta herramienta, SDSoC es una herramienta que puede usar cualquiera que sepa programar en C/C++, pero cuando el sistema a implementar no requiere de una plataforma hardware asociada. Si el sistema que se quiere hacer es más completo, se requerirá de un ingeniero de hardware que desarrolle la plataforma en Vivado.

Además, implementar sistemas con esta herramienta es un punto bastante a su favor, en cuanto al tiempo de desarrollo para salga dicho sistema al mercado se refiere, así se pueden dar soluciones en menos tiempo que la competencia.

Por otra parte, este proyecto puede tener bastantes vías futuras de desarrollo, que iré describiendo a continuación.

El problema al que se enfrenta esta plataforma, es que la FPGA para la que está implementada es bastante pequeña. Lo propio sería adaptar este sistema a una FPGA donde pudiéramos sintetizar más de un filtro para el hardware y así tener más opciones a elegir. Lo ideal sería usar una FPGA de la misma familia (Zynq 7000) pero con mayor número de recursos [25].

Otra vía de desarrollo interesante, es que el sistema en vez de ser *standalone* se montara sobre un Linux embebido [22] [23]. Así tendríamos soporte de un SO, teniendo muchos servicios a nuestra disposición. Además de poder usar varios procesos en paralelo haciendo uso de los dos procesadores ARM que hay en el SoC. Sobre Linux también se tendría la opción de poder hacer una reconfiguración dinámica de la FPGA, pero sería necesaria una FPGA de mayor tamaño.

7. Bibliografía final

[1] Plataforma Vivado:
https://github.com/Digilent/SDSoC-platforms/tree/master/2015.4/zybo_hdmi_in

[2] Arquitectura familia Xilinx Zynq-7000:
https://www.digikey.com/en/product-highlight/x/xilinx/zynq-7-soc

[3] Imagen Zynq XC7Z020-2CLG400I:
https://media.digikey.com/Photos/Xilinx%20Photos/XC7Z020-2CLG400I.jpg

[4] Arquitectura Zynq Ultrascale:
https://xlnx.i.lithium.com/t5/image/serverpage/image-id/20862i691BDD6A07B57E4B?v=1.0

[5] Herramienta SDAccel:
https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html

[6] Documentación placa de desarrollo Zybo:
https://reference.digilentinc.com/_media/zybo:zybo_rm.pdf

[7] Herramienta SDSoC:
https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html

[8] Herramienta Vivado HLS:
https://www.xilinx.com/products/design-tools/hardware-zone.html

[9] Herramienta Vivado:
https://www.xilinx.com/products/design-tools/vivado.html

[10] Documentación Vivado HLS:
https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[11] Manual Vivado HLS:
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug902-vivado-high-level-synthesis.pdf

[12] SoCs de Intel:
https://www.altera.com/products/soc/overview.html

[13] SoC Stratix 10:
https://www.altera.com/products/fpga/stratix-series/stratix-10/features.html

[14] SoC Arria 10:
https://www.altera.com/products/soc/portfolio/arria-10-soc/features.html

[15] SoC Arria V:
https://www.altera.com/products/soc/portfolio/arria-v-soc/features.html

[16] Filtro para procesamiento de imagen con Vivado HLS:
https://www.xilinx.com/support/documentation/application_notes/xapp890-zynq-sobel-vivado-hls.pdf

[17] Documentación VDMA:
https://www.xilinx.com/products/intellectual-property/axi_video_dma.html

[18] Documentación AXI4:
https://www.xilinx.com/products/intellectual-property/video_in_to_axi4_stream.html

[19] Codificador JPEG para Linux:
https://github.com/r-lyeh/moon9/blob/master/deps/render/jojpeg/jo_jpeg.cpp

[20] Biblioteca Xilffs:
https://github.com/Xilinx/embeddedsw/blob/master/lib/sw_services/xilffs/src/include/ff.h

[21] Guía de usuario SDSoC:
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug1027-sdsoc-user-guide.pdf

[22] Sistema operativo Xilinux:
http://xillybus.com/xillinux

[23] Herramienta de compilación de kernel Petalinux:
http://www.wiki.xilinx.com/PetaLinux

[24] Libro, FPGAs: Instant Access, Clive Max Maxfield, 2008.

[25] Familia de SoCs Xilinx familia Zynq-7000:
https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html#productTable

[26] W. Burger, Mark J. Burge, "Digital Image Processing: An Algorithmic Introduction Using Java", 2nd. ed., Springer, 2016.

[27] John C. Russ, F. Brent Neal, "The image processing handbook", Seventh edition, CRC Press, 2016.

[28] G. Bueno Garcia, et al., "Learning image processing with OpenCV : exploit the amazing features of OpenCV to create powerful image processing applications through easy-to-follow examples", Pakct Publishing, 2015.

ANEXO A. Actualización de la plataforma hardware a una versión actual.

Siguiendo la línea de trabajo sobre la parte hardware de la plataforma, vamos a ver cómo podemos actualizar el proyecto de una versión anticuada a una versión más actual.

El primer paso es abrir el proyecto Vivado. Estos proyectos tienen una extensión .xpr, donde podemos abrirlos haciendo doble clic sobre ellos o buscando el proyecto desde la interfaz de Vivado.

Una vez arranque el proyecto, nos saltará un aviso. Que nos indicará que el proyecto procede de una versión más antigua y que se recomienda actualizar para poder ser utilizado debidamente. Dejamos seleccionado que se actualice automáticamente y le damos a OK.

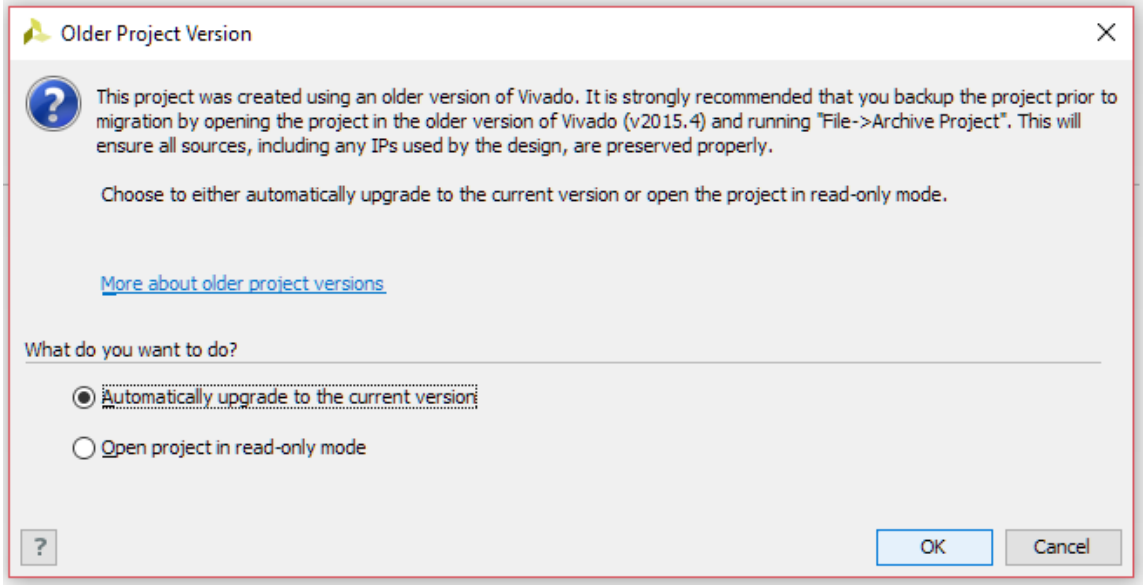


Figura 29: Actualizar plataforma.

La herramienta nos avisará que tenemos que revisar los core IP, porque puede haber cambios en la implementación de una versión a otra. Pulsamos el botón “Report IP Status”.

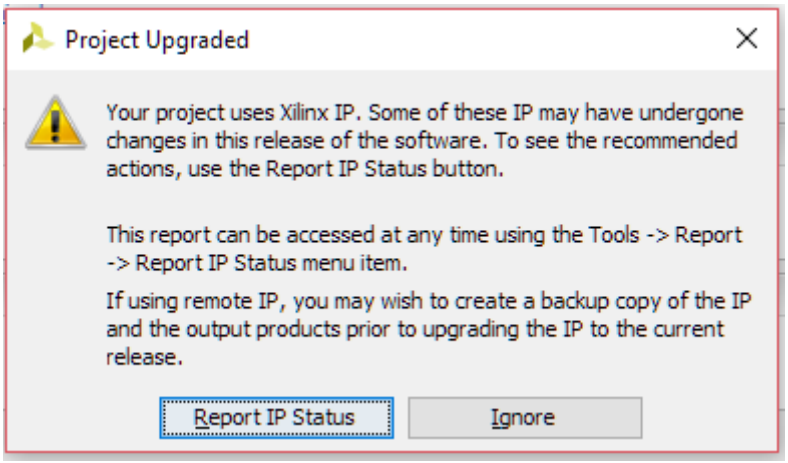


Figura 30: Actualizar cores IP

Que nos conducirá a una ventana donde tenemos una lista con todos los core IP que necesitan ser revisados. Hacemos clic en el botón “Upgrade Selected”

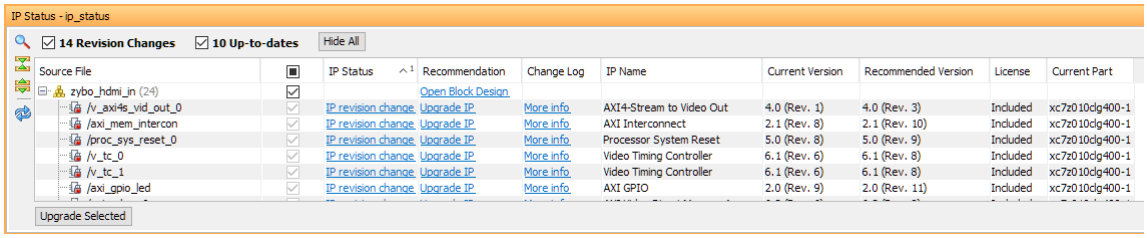


Figura 31: Actualizar seleccionados.

La herramienta actualizará todos los componentes de la lista. Avisándonos de que se ha completado el proceso correctamente o de que no lo ha hecho.
En caso de que se haya completado el proceso correctamente, nos saldrá una ventana dándonos la confirmación. Que en caso de pulsar el “ok” nos mandará a otra ventana donde ya tendremos la posibilidad de sintetizar e implementar la plataforma.

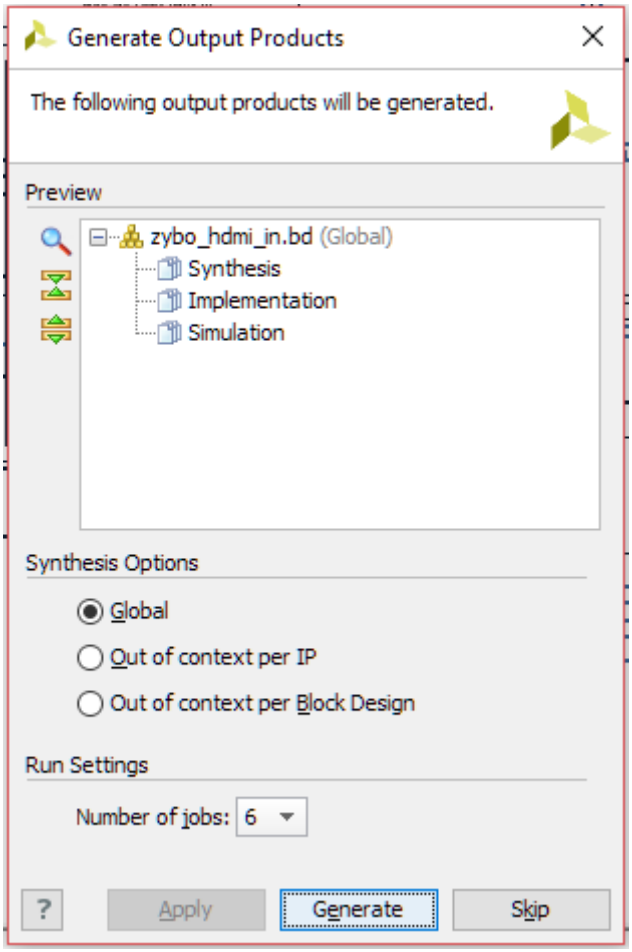


Figura 32: Implementar plataforma

Pulsamos sobre “Generate” y Vivado se pondrá a trabajar.

Una vez realizado todo este procedimiento, solo tendríamos que exportar el hardware para que tengamos el archivo disponible. Que será requerido por SDSoC cuando vayamos a crear la aplicación.