



*ugr* | Universidad  
de Granada



## PROCESAMIENTO DE IMÁGENES EN TIEMPO REAL PARA SISTEMAS EMPOTRADOS CON FPGA

PROYECTO FIN DE CARRERA

Curso 2015/2016

Ingeniería Superior en Telecomunicaciones

Manuel Santiago Vargas Plata





*ugr*

Universidad  
de Granada



## INGENIERÍA SUPERIOR EN TELECOMUNICACIONES

Procesamiento de imagen en tiempo real para sistemas empotrados con  
FPGA

REALIZADO POR:

Manuel Santiago Vargas Plata

DIRIGIDO POR:

Javier Díaz Alonso

Francisco Barranco Expósito

DEPARTAMENTO:

Arquitectura y Tecnología de Computadores

Granada, Diciembre del 2015

Procesamiento de imagen en tiempo real para sistemas empotrados con FPGA

Manuel Santiago Vargas Plata

**Palabras clave:** Procesamiento imagen, FPGA, vídeo, tiempo real, altas prestaciones, Zybo, Zynq, sistemas empotrados, Vivado, HLS, VHDL, VDMA, HDMI, VGA.

**Resumen:** El objetivo de este proyecto, es demostrar el potencial de las nuevas arquitecturas System on Chip (SoC) con FPGA, para procesamiento eficiente de imágenes en sistemas empotrados. Se lleva a cabo un filtrado de Sobel paralelizado para la extracción de bordes de una imagen en alta resolución (1280x720 píxeles). Todo el proceso de filtrado, se realiza a través de un co-procesamiento ejecutado en una FPGA capaz de reprogramarse, y que proporciona una gran potencia de cálculo y gran rendimiento. Se utiliza la tecnología Vivado HLS para programar hardware en alto nivel, siendo esta herramienta la encargada de traducir el software de alto nivel, a programación de bajo nivel, obteniendo con ello los mejores y más eficientes resultados de una manera sencilla para el diseñador. Además se usa la herramienta Vivado para generación de la arquitectura que, gracias a su diseño por bloques con una interfaz gráfica, cómoda, rápida y eficiente.

**Keywords:** Image processing, FPGA, vídeo, real time, high performance, Zybo, Zynq, Embedded systems, Vivado, HLS, VHDL, VDMA, HDMI, VGA.

**Abstract:** The target of this project, is to prove the potential of the new System on Chip (SoC) architectures with FPGA, for efficient image processing in embedded systems. It is performed a parallelized Sobel filter for extracting edges if a high resolution image (1280x720 pixels). All the filtering process is performed through a co-processing executed on a FPGA which can be reprogrammed and that provides a high computing power and high performance. The Vivado HLS hardware technology is used to program high level code, being in charge of translating the high-level software to a low-level program code, obtaining the best and most efficient results in a simple way for the designer. Also, the Vivado tool is used to generate the architecture thanks to its block design with a graphic, handy, fast and efficient.



D. Javier Díaz Alonso, Profesor del departamento de Arquitectura y Tecnología de Computadores de la Escuela Técnica Superior de Ingenierías de Informática y de Telecomunicación de la Universidad de Granada, como director del Proyecto Fin de carrera de D. Manuel Santiago Vargas Plata.

Informa:

que el presente trabajo, titulado:

"Procesamiento de imagen en tiempo real para sistemas empotrados con FPGA"

ha sido realizado y redactado por el mencionado alumno bajo mi dirección, y con esta fecha autorizo a su presentación.

Granada, a 17 de Diciembre de 2015

Fdo. Javier Díaz Alonso



Los abajo firmantes autorizan a que la presente copia de Proyecto Fin de Carrera se ubique en la Biblioteca del Centro y/o departamento para ser libremente consultada por las personas que lo deseen.

Granada, a 17 de Diciembre de 2015

Fdo. Manuel Santiago Vargas Plata

DNI: 75569697-T

Fdo. Javier Díaz Alonso

DNI:

Fdo. Francisco Barranco Expósito

DNI:



# Agradecimientos

---

Tras 5 años de carrera, y más de otro año para realizar este proyecto fin de carrera, tengo mucho que agradecer a tantas, y diversas personas que han aportado algo a todos estos años de esfuerzo y dedicación a la ingeniería.

Tengo que comenzar por agradecer a mis padres su apoyo incondicional durante todos mis años de carrera, desde que elegí estudiar ingeniería en telecomunicaciones hasta que he llegado hasta aquí. Sin ellos, no habría llegado hasta este punto, tanto por los ánimos en momentos de flaqueza, como por pagarme los estudios.

Por supuesto, tengo que agradecer también a mis dos hermanas mayores, que siempre han estado ahí cuando las he necesitado, sin importar para qué. A la primera, por ayudarme y encarrilarme en el mundo de la ingeniería, y a la segunda, por ayudarme a ver la vida más allá de la carrera.

A mis amigos y amigas de toda la vida, esos con los que desde la adolescencia he compartido buenos y no tan buenos momentos, pero siempre, momentos para recordar con alegría.

No puedo olvidar a mis compañeros de carrera, amistad que siempre nos unirá, estemos donde estemos en el país o el planeta, sin los cuales no habría sido capaz de afrontar toda la carrera.

Por último, y no por ello menos importante, a mi pareja. La cual lleva en este momento casi 4 años aguantándose los llantos acaecidos tanto por la carrera como por el trabajo y a la cual tengo mucho que agradecer.

A todos, mi más sincero agradecimiento.

Manuel Santiago Vargas Plata.



# Índice de Contenidos

---

Siglas y definiciones.....	17
1. Introducción.....	18
1.1 Descripción del problema.....	18
1.2 Motivación.....	20
1.3 Objetivos.....	23
1.4 Posibles aplicaciones y diseños similares .....	24
1.4.1 Separar materiales en una cadena de montaje .....	24
1.4.2 Identificación de matrículas .....	25
1.5 Materiales y métodos.....	26
1.5.1 Materiales.....	26
1.5.1.1 Ordenador portátil.....	26
1.5.1.2 Placa de desarrollo Zybo Zynq – 7000. ....	26
1.5.1.3 Monitor VGA. ....	27
1.5.2 Métodos.....	27
1.5.2.1 Vivado. ....	27
1.5.2.2 Xilinx SDK.....	32
1.5.2.3 Vivado HLS.....	37
1.6 Marco de trabajo. ....	40
1.6.1 Proyecto VITVIR .....	40
1.6.2 Proyecto MULTIVISION.....	41
1.7 Estructura de la memoria. ....	42
2 Sistemas de visión empotrada.....	42
3. Arquitecturas y diseños .....	44
3.1 Diseños de referencia .....	44
3.1.2 Zybo Base Design .....	44
3.1.3 Image Filtering Demo + GoPRo.....	50
3.2 Diseño propio. Procesado por software con 2 VDMAs. ....	53
3.3 Diseño propio. Procesado por hardware con 3 VDMAs. ....	58
3.3.1 Programación y diseño del Core HLS de procesamiento de vídeo.....	63
3.3.2 Configurando los VDMA .....	74
3.3.3 Consumo de recursos y compromiso de diseño .....	83
4. Conclusiones. ....	86
5. Bibliografía .....	89

ANEXO A. Detalle de los distintos cores usados.....	91
1. Core HDMI_RX .....	91
1.1 Entradas.....	91
1.2 Salidas: .....	91
2. Core AXI Display Controller .....	92
2.1 Entradas:.....	92
2.2 Salidas .....	93
3. Core Vídeo In to AXI4-Stream.....	93
2.1 Entradas.....	93
2.2 Salidas .....	94
4. Core VDMA. ....	94
4.1 Entradas.....	95
4.2 Salidas .....	96
5. Core Constant .....	97
6. Core AXI Interconnect.....	98
6.1 Entradas.....	100
6.2 Salidas .....	100
7 AXI Protocol Converter .....	101
7.1 Entradas.....	101
7.2 Salidas .....	102
8 Core Utility Vector Logic.....	102
8.1 Entradas.....	103
8.2 Salidas .....	103
9 Core AXI GPIO .....	103
9.1 Entradas:.....	104
9.2 Salidas: .....	104
10 Core ZYNQ7 Processing System.....	105
10.1 Entradas.....	106
10.2 Salidas .....	106

# Índice de ilustraciones

Ilustración 1. Representación de escaneo progresivo.....	23
Ilustración 2. Ejemplo visión tornillos/tuercas .....	25
Ilustración 3. Imagen tratada con detección de bordes.....	25
Ilustración 4. <sup>[SHO,15]</sup> Fotografía de la placa de desarrollo Zybo Zynq 7000 .....	27
Ilustración 5. <sup>[DIG, 15, B]</sup> Logo Vivado .....	28
Ilustración 6. Interfaz gráfica de Vivado: Diseño de bloques a la derecha.....	28
Ilustración 7. Indicación de insertar IP Core en diseño vacío .....	29
Ilustración 8. Ejemplo de conexión automática 1 .....	29
Ilustración 9. Ejemplo de conexión automática 2 .....	30
Ilustración 10. Ejemplo de líneas marcadas para debug .....	30
Ilustración 11. Pasos para configurar Debug .....	31
Ilustración 12. Ejemplo ChipScope .....	31
Ilustración 13. Exportar hardware (1).....	32
Ilustración 14. Exportar hardware (2).....	32
Ilustración 15. Lanzar el SDK desde Vivado .....	33
Ilustración 16. Interfaz gráfica de Xilinx SDK .....	33
Ilustración 17. Hardware platform creado por Xilinx SDK .....	34
Ilustración 18. BSP con todas las librerías que incluye.....	34
Ilustración 19. Proyecto con todos los ficheros de código C .....	35
Ilustración 20. Conexión Zybo - PC mediante cable microUSB.....	35
Ilustración 21. Botón de descarga del bitstream.....	35
Ilustración 22. Placa con bitstream cargado.....	36
Ilustración 23. Detalle del LED LD10 "DONE" iluminado .....	36
Ilustración 24. Ejecutar proyecto en la placa de desarrollo .....	36
Ilustración 25. Logo Vivado HLS.....	37
Ilustración 26. Interfaz gráfica de Vivado HLS .....	37
Ilustración 27. Detalle de pestaña "Directive".....	38
Ilustración 28. Detalle menú "Insert directive..." .....	38
Ilustración 29. Ejemplo de informe de señales Vivado HLS.....	39
Ilustración 30. Aviso de Vivado de que hay un core sin actualizar .....	39
Ilustración 31. Ejemplo del "Report IP Status" de Vivado .....	40
Ilustración 32. Diseño de partida: Zybo Base Design.....	45
Ilustración 33. Detalle del diseño base (1).....	46
Ilustración 34. Detalle del diseño base (2).....	47
Ilustración 35. Detalle del diseño base (3).....	48
Ilustración 36. Core IP HDMI RX .....	50
Ilustración 37. Conexionado para puente HDMI-VGA.....	50
Ilustración 38. Detalle del diseño por software (1) .....	54
Ilustración 39. Detalle del diseño por software (2) .....	55
Ilustración 40. Detalle del diseño por Hardware (1).....	59
Ilustración 41. Detalle del diseño por Hardware (2).....	60
Ilustración 42. Detalle del diseño por Hardware (3).....	61
Ilustración 43. Ventana principal Vivado HLS .....	67
Ilustración 44. Iniciar síntesis de la solución.....	68
Ilustración 45. Directorio SYN creado tras la síntesis .....	68
Ilustración 46. Resumen de tiempos de ejecución .....	69
Ilustración 47. Resumen y detalle de la latencia en la ejecución .....	69
Ilustración 48. Resumen de la utilización .....	69

Ilustración 49. Detalle de utilización de la instancia .....	70
Ilustración 50. Detalle de la utilización FIFO.....	70
Ilustración 51. Detalle de la utilización de la expresión .....	70
Ilustración 52. Detalle de la utilización del multiplexor .....	71
Ilustración 53. Detalle de utilización de los registros .....	71
Ilustración 54. Informe de las interfaces a generar.....	72
Ilustración 55. Exportar el RTL.....	73
Ilustración 56. Directorio "impl" y Core a importar en Vivado.....	73
Ilustración 57. Crear un BSP (1) .....	74
Ilustración 58. Crear un BSP (2) .....	75
Ilustración 59. System.mss del BSP generado .....	75
Ilustración 60. Configuración del VDMA HDMI en Vivado .....	79
Ilustración 61. Configuración del VDMA HLS en Vivado.....	82
Ilustración 62. Configuración del VDMA VGA en Vivado.....	83
Ilustración 63. Datos de temporización (1) .....	83
Ilustración 64. Datos de temporización (2) .....	84
Ilustración 65. Datos de temporización (3) .....	85
Ilustración 66. Utilización de recursos del diseño .....	85
Ilustración 67. Potencia de consumo del diseño .....	86
Ilustración 68. Detalle del core HDMI_RX. ....	91
Ilustración 69. Salidas y entradas core Display Controller .....	92
Ilustración 70. Entradas y salidas del core Vídeo In To Axi4-Stream.....	93
Ilustración 71. Entradas y salidas del core AXI Vídeo Direct Memory Access con lectura/escritura .....	95
Ilustración 72. Entrada S_AXIS_S2MM en detalle. ....	95
Ilustración 73. Salida M_AXIS_MM2S en detalle.....	96
Ilustración 74. Detalle del core constant .....	97
Ilustración 75. Ventana de configuración del core Constant .....	97
Ilustración 76. Diagrama de conexiones internas del AXI Interconnect [XIL,15,E] Figura 2-1 .....	98
Ilustración 77. Interconexión realizara por el AXI Interconnect tipo N-to-1. [XIL,15,E] Figura 2-2.....	98
Ilustración 78. Interconexión realizada por el AXI Interconnect tipo 1-to-N. [XIL,15,E] Figura 2-3.....	99
Ilustración 79. AXI Interconnect tipo N-to-1.....	99
Ilustración 80. AXI Interconnect tipo 1-to-N.....	99
Ilustración 81. Detalle interior del AXI Interconnnect 1-to-N.....	100
Ilustración 82. Core AXI Protocol Converter.....	101
Ilustración 83. Configuración del core AXI Protocol Converter.....	101
Ilustración 84. Conexión procesador - AXI Interconnect .....	102
Ilustración 85. Core Utility Vector Logic .....	102
Ilustración 86. Configuración del core Utility Vector Logic .....	103
Ilustración 87. Core AXI GPIO .....	103
Ilustración 88. Crear señal externa en AXI GPIO .....	104
Ilustración 89. Señal externa creada .....	105
Ilustración 90. Core ZYNQ7 Processing System .....	105
Ilustración 91. Interfaz de configuración del core ZYNQ7 Processing System .....	106
Ilustración 92. Configuración de los relojes del ZYNQ7 PRocessing System .....	107

## Índice de códigos

---

Código 1. Definición puertos SINCRONISMO.....	51
Código 2. Definición puertos canal AZUL.....	51
Código 3. Definición puertos canal ROJO. ....	51
Código 4. Definición puertos canal VERDE. ....	51
Código 5. Copiado mediante bucle for .....	57
Código 6. Copiado mediante "memCpy" .....	58
Código 7. Core HLS Sobel (1). Directrices para interfaces .....	65
Código 8. Core HLS Sobel (2). Declaración de variables intermedias necesarias .....	65
Código 9. Core HLS Sobel (3). Distintas funciones de filtrado de imagen .....	65
Código 10. "Sobel.h" incluido en el TOP del core .....	66
Código 11. Configuración del VDMA HDMI .....	77
Código 12. Configuración del VDMA HLS (Lectura y escritura) .....	81
Código 13. Configuración de los switches .....	105

## Índice de esquemas

---

Esquema 1. Diseño por software 1.....	56
Esquema 2. Diseño por software 2.....	57
Esquema 3. Diseño por hardware .....	62
Esquema 4. Esquema diseño VDMAs .....	80

## Siglas y definiciones

- ■ ■ **ASIC:** Application-Specific Integrated Circuit (Circuito integrado para aplicaciones específicas).
- ■ ■ **BSP:** Board Support Package (Paquete de soporte de la placa). Se refiere a todos los componentes software necesarios para un diseño hardware determinado.
- ■ ■ **GPU:** Graphics Processing Unit (Unidad de procesamiento gráfico).
- ■ ■ **DSP:** Digital Signal Processor (Procesador de señales digitales).
- ■ ■ **FPGA:** Field Programmable Gate Array (“Campo de matriz de puertas programables”).
- ■ ■ **Resolución HD:** Dimensiones de imagen de 1280 x 720 (ancho x alto) píxeles.
- ■ ■ **Resolución Full HD:** Dimensiones de imagen de: 1920 x 1080 (ancho x alto) píxeles.
- ■ ■ [XAT, 15] **Resolución 2K:** Dimensiones de imagen de: 2048 x 1080 (ancho x alto) píxeles.
- ■ ■ [XAT, 15] **Resolución 4K:** Dimensiones de imagen de: 4096 x 2160 (ancho x alto) píxeles.
- ■ ■ **SDK:** software development kit (Kit de desarrollo software).
- ■ ■ **TOF:** Time Of Flight
- ■ ■ **VDMA:** **Vídeo Direct Memory Access:** (Acceso directo a la memoria de vídeo).

## 1. Introducción

---

### 1.1 Descripción del problema.

El diseño de sistemas embebidos ha sido siempre un gran campo de trabajo para los ingenieros hardware. En estos diseños, en los que se busca el mayor rendimiento posible, era necesario programar en un lenguaje bajo nivel, como VHDL o ensamblador. Esto resulta engorroso, complejo y conlleva grandes ciclos de trabajo, lo que encarece los productos pues retrasa su salida al mercado y son necesarias muchas más horas de desarrollo.

Sin embargo, con la salida al mercado de los nuevos sistemas de desarrollo, cada vez se simplifica más el proceso de creación de sistemas hardware específicos.

Es por ello, que en este proyecto se quiere estudiar el rendimiento de los nuevos sistemas de desarrollo, en particular, una placa de desarrollo fabricada por Xilinx: La Zybo Zynq 7000. Ésta cuenta con un procesador doble núcleo y una FPGA reprogramable, para poder realizar todas las pruebas necesarias, hasta obtener el producto final deseado para poder enviarlo a fabricar en cadena, consiguiendo reducir costes y aumentar la productividad.

Además, se quieren probar y demostrar la utilidad de las herramientas de Xilinx: Vivado y Vivado HLS para el desarrollo hardware. La automatización que éstas brindan, permiten un diseño hardware mucho más rápido que años atrás, permitiendo además que los ingenieros software puedan realizar importantes dispositivos hardware.

Para comprobar el funcionamiento, tanto del software de diseño de Xilinx, como de la placa de desarrollo Zybo Zynq-7000, se decide realizar una aplicación de captura, procesamiento y muestra de vídeo en tiempo real con el objetivo de poder exprimir las características de la placa y demostrar que, con una pequeña placa de desarrollo como es la Zybo, se puede conseguir gran potencia de procesamiento.

En parte, el querer realizar un diseño de procesado de vídeo en tiempo real, se debe a mi afición a la robótica y a los sistemas autónomos, donde podrían integrarse este tipo de sistemas de procesado de vídeo, pues puede ser muy útil en aplicaciones comerciales como:

- Separación de artículos en una cadena de montaje en función del tamaño, forma o color, de manera automática.
- Mejorar la fluidez en el proceso de verificación de artículos permitidos en el acceso a un recinto controlado, como puede ser un aeropuerto, un estadio o una fábrica industrial.
- Control de sistemas informáticos con gestos en el aire.
- Ayuda a sistemas de visión artificial para personas con visibilidad reducida.
- El tan mencionado actualmente coche autónomo.

 Y un larguísimo etcétera.

Un brazo robótico debe ser capaz de interpretar lo que tiene delante, para actuar en consecuencia. Milésimas de segundo en el procesado de la imagen pueden ser críticas. Es por ello, que deben usarse dispositivos capaces de realizar el cálculo de las instrucciones de acción de forma rápida y efectiva, así como de un diseño software-hardware eficiente y bien estructurado.

Tiempo atrás, el procesado de imagen en tiempo real era un complicado trabajo de investigación. No obstante, en la actualidad los nuevos sistemas que tenemos en el mercado, hacen que esta tarea sea mucho más sencilla y eficiente. Las aplicaciones para procesamiento de imagen, diseñadas por software, producen numerosos cuellos de botella ya que este procesado es muy intensivo y no es sencillo cumplir los requisitos de tiempo real.

Existen varias opciones para tratar este tema:

 [WIK, 15, C] **Sistemas ASIC**: Se trata de circuitos integrados para aplicaciones específicas (*Application-Specific Integrated Circuit*) diseñados en particular para realizar una tarea. El gran problema de estos sistemas es que deben diseñarse a muy bajo nivel. Además, una vez fabricado, sólo podrá utilizarse con el fin con el que se diseñó, por lo tanto, no es nada escalable, ni vale para poder realizar diferentes diseños para así comprobar su eficiencia. Son útiles cuando se realizan con propósito de producción en masa, como por ejemplo un chip para controlar la cámara de un teléfono móvil.

 [WIK, 15, E] **Sistemas tipo GPU**: Hablamos de una unidad de procesamiento gráfico (*Graphics Processing Unit*) que se dedica exclusivamente a esto. Sin embargo, el consumo de potencia es elevado en comparación con otros sistemas. Estas GPUs suelen ir incorporadas en los PCs de sobremesa, para dar un apoyo en el procesamiento gráfico a la CPU, para así liberarla de carga que pueda dedicarse a otras tareas.

 [WIK, 15, F] **Sistemas DSP**: Este caso, son los procesadores digitales de señales (*Digital Signal Processor*), sistemas basados en un procesador o microprocesador que con un conjunto de instrucciones y un hardware-software optimizados, se dedican a aplicaciones que requieran operaciones numéricas a altas velocidades. Es común utilizar estos sistemas junto con conversores analógico/digitales para trabajar en tiempo real. Aunque bien podría servir para el tema que queremos tratar aquí, sus bajas prestaciones los hacen insuficientes para nuestra tarea.

 [WIK, 15, G] **Sistemas FPGA**: “Campo de matriz de puertas programables” o mejor, su nombre original: *Field Programmable Gate Array* se trata de un dispositivo que contiene bloques de lógica, que pueden ser programados de forma que la conexión entre ellos se pueda modificar tantas veces como sea necesario, a través de un software específico. Aunque estos sistemas sean algo más lentos

que los ASIC y tengan un consumo algo más elevado, cuentan con la gran ventaja de poder ser reprogramadas tantas veces como sean necesarias, hasta conseguir el diseño y las prestaciones requeridas. Además, son sistemas mucho más baratos, flexibles en el flujo de diseño y el tiempo de desarrollo es reducido.

Por las razones que se especifican de los sistemas FPGA, es por lo que se ha elegido como el sistema clave para poder realizar este proyecto, pues contamos con un dispositivo asequible, tanto económicamente hablando, como a la hora de programar el diseño.

Día a día surgen nuevos dispositivos capaces de trabajar mejor, más rápido y con más eficiencia. No obstante, a la par que mejora la tecnología de procesamiento también lo hace la tecnología de captura del vídeo. Resoluciones HD, FullHD, 2K, 4K y en aumento, precisan cada vez de más rendimiento y carga de trabajo, siendo los sistemas que realizan el tratamiento directamente en el procesador, incapaces de trabajar con tal cantidad de información en tiempo real. Es por esto que los sistemas FPGA son los más recomendados para ello, ya que aunque se modifique su hardware reprogramable para acelerar distintos procesos, el sistema operativo que la controla no recibe cambios

No es un tema sencillo trabajar en un diseño hardware-software y su interacción con distintos periféricos. Sin embargo, existen herramientas que ayudan al programador en la ardua tarea del diseño hardware, con la opción posterior de indagar en su proyecto y conseguir un rendimiento aún mayor conforme baja el nivel de abstracción al programar.

Este es el concepto de *Codiseño* en el que podemos desarrollar sistemas de procesamiento de imágenes en tiempo real, en los que la parte más compleja del procesado, se realiza en un hardware diseñado específicamente para esto, pero dentro de la misma FPGA. Además, este codiseño se realiza mediante lenguajes de síntesis de alto nivel, consiguiendo minimizar los ciclos de diseño hardware y con ello, mejorar la depuración y realizar proyectos de esta índole, mucho más complejos de los que permiten lenguajes de programación HDL

Es por todo esto que este proyecto, está dedicado a la creación de sistemas de procesado de vídeo en tiempo real, usando para ello materiales asequibles, relativamente sencillos de utilizar y herramientas de diseño hardware y software que permiten al programador una integración asistida más interactiva e intuitiva para sus proyectos.

## 1.2 Motivación.

Desde que empecé a pensar en que quería dedicarme a la ingeniería, siempre he sido un gran amante del tema de robótica. Cuando comencé la carrera de Ingeniería en Telecomunicaciones encontré asignaturas que ofertaban programas orientados a temas relacionados con estos seres “automáticos”, asignaturas orientadas a inteligencia artificial, procesado de imagen, programación hardware, etc...

Es por esto que finalmente me decanté por realizar un proyecto fin de carrera que estuviera relacionado con una parte de los robots, que es la visión artificial.

Realizar un sistema que permita obtener imagen en tiempo real y realizar un tratamiento de la misma con un objetivo predefinido, es a mi parecer uno de los pasos previos que se deben realizar para la creación de un ente automático, entendiendo como robot, cualquier máquina que pueda realizar una tarea más, o menos complicada, de forma automática y sin supervisión.

Sin embargo, la visión artificial es utilizada desde un sencillo gancho robótico que se encargue de separar “pelotas azules” de “pelotas rojas”, hasta un humanoide capaz de realizar tareas complejas que precisen captar todo el entorno por el que se mueva.

De hecho, uno de los sistemas más sencillos que vi antes de decantarme por realizar un proyecto basado en visión artificial, fue un sencillo diseño que separaba los caramelos M&M's por colores.

Una vez que decidí realizar un proyecto de visión artificial, comencé a investigar cómo hacerlo y es como llegué a decidir, con ayuda de mi tutor, que quería realizar un proyecto de captura y tratamiento de imagen, programando hardware.

La decisión de realizarlo utilizando una Zybo Zynq y el software de desarrollo Vivado, es debido a su interfaz sencilla, que ayuda al diseño hardware gracias a una programación mediante bloques que permite la configuración de éstos directamente realizando un doble clic, y a través de una interfaz gráfica intuitiva y sencilla.

Además, esta pequeña placa de desarrollo, contiene todas las interfaces necesarias para el objetivo de este proyecto:

- Puerto HDMI para entrada.
- Puerto VGA para salida.
- Switches y botones que pueden utilizarse de la forma que se desee.
- Programación sencilla a través de un puerto MicroUSB.
- Alimentación externa mediante transformador y enchufe a la pared.
- Lector de tarjetas microSD que podría utilizarse para cargar el programa para un diseño stand-alone

Pero, ¿por qué realizar una arquitectura con VDMAs en lugar de hacer directamente una lectura desde memoria?

Ésta es una pregunta que puede parecer obvia, ya que sería muy sencillo realizar la lectura de un vídeo o una imagen directamente desde la memoria de la placa, aplicarle el procesado y mostrarlo. Todo esto directamente realizando lecturas en la propia memoria.

Pero con un diseño así no se conseguiría lo más importante que se busca en este proyecto, que es el procesado de vídeo **en tiempo real**. El gran añadido de esta arquitectura es el de poder capturar vídeo en tiempo real, y simultáneamente, aplicarle un procesado que luego pueda utilizarse para distintas funciones.

Además, la arquitectura final en la que se incluyen 3 VDMAs y un core de procesado de vídeo, realizado con Vivado HLS, permite que otro programador en el futuro, sea capaz, a través de sencillas **modificaciones software**, de aplicar un procesado al vídeo sin tener que entrar de lleno a programación hardware, mucho más engorrosa y compleja. Es lo que se conoce como **reutilización**.

Normalmente, este tipo de procedimientos se reservan a ingenieros hardware que están más acostumbrados al trabajo con estos diseños. Sin embargo, gracias a las herramientas de Xilinx, Vivado y Vivado HLS, se pueden conseguir grandes diseños de hardware en alto nivel, que más adelante la misma herramienta será capaz de traducir al lenguaje hardware de bajo nivel.

Bajo otro punto de vista, realizar el diseño con un VDMA dedicado a la entrada de video y otro a la salida del mismo, ofrece la posibilidad al desarrollador de cambiar las interfaces de entrada y salida sin tener que modificar todo el cableado interno. Es decir, eliminando el core de captura de vídeo HDMI, y el core de salida de vídeo VGA, se podrían incluir otros diferentes para otras interfaces distintas de vídeo y volver a implementar, sin preocuparse del resto de componentes.

Otra pregunta puede ser, ¿Y por qué usar una entrada HDMI? Desde un puerto HDMI, se transporta tanto vídeo como audio **digital** es decir, se trata de un puerto que permite la recepción de vídeo de **alta resolución digital** y audio **digital** con **un alto bitrate de transferencia** consiguiendo por tanto una mejor calidad de vídeo que otros sistemas, como el VGA, el S-Vídeo o similares.

Como en este diseño, no se va a tratar la captura del audio, sino sólo del vídeo ya que es el tema importante a la hora de procesar y actuar en función de lo que se ve. Del puerto HDMI se utiliza sólo el vídeo, que de nuevo, recordemos que se obtiene vídeo de **alta definición y digital**.

Pero... ¿Y entonces por qué se utiliza como salida un puerto VGA? La principal razón son las limitaciones de la placa, ya que sólo contiene como puertos de entrada/salida de vídeo, un puerto HDMI y un puerto VGA.

El puerto HDMI puede configurarse para ser usado como una fuente, o un sumidero, es decir, una entrada o una salida para vídeo.

El puerto VGA sin embargo, sólo puede utilizarse como sumidero, es decir, como salida de vídeo. Por tanto, se decide utilizar el puerto HDMI como entrada, ya que, aparte de que nos proporciona una imagen de vídeo en alta definición, existen gran número de dispositivos que pueden utilizarse como entrada con este tipo de interfaz, como videocámaras, webcams, videoconsolas, móviles, ordenadores...

Y pensando... ¿Qué equipo utiliza un VGA como salida de vídeo que podamos recoger? Pues, prácticamente, ninguno. Es por ello que se decide utilizar esta arquitectura.

Por otra parte, el puerto VGA de la placa, permite salida de vídeo en alta resolución, aunque en formato analógico, pero para este diseño, es más que suficiente.

La Zybo Zynq es capaz de obtener imagen desde el HDMI en una resolución máxima probada por Xilinx, de 1280x720 píxeles la denominada resolución “720p”. La “p” indica que es un escaneo progresivo, es decir, el barrido de la imagen se realiza en líneas sucesivas:

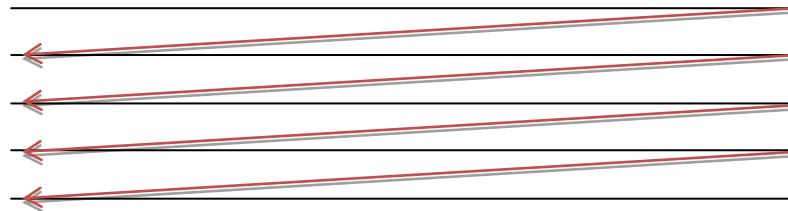


Ilustración 1. Representación de escaneo progresivo

Con lo que, incluso televisores de 40 ó 42 pulgadas se ven con una imagen nítida y sin pixelar a esta resolución.

### 1.3 Objetivos

El objetivo de este proyecto fin de carrera es el de estudiar el comportamiento de un sistema empotrado, a la hora de procesar una imagen en tiempo real, y demostrar el gran potencial de los sistemas embebidos y sus posibilidades de uso.

De una forma sencilla, se quiere obtener una imagen en directo a la cual, se le aplique un procesamiento, más o menos sencillo, de manera que la imagen resultante sea la obtenida en tiempo real con ese procesado realizado a la par con la captura.

Para ello se hará uso de una placa de desarrollo: **Zybo Zynq-7000 de Digilent**, la cual es una placa de pequeñas dimensiones y coste reducido, pero con gran potencia de cálculo y procesado, gracias a su procesador de doble núcleo, y su FPGA integrada, equivalente a una Artix-7.

El sistema se completa con un dispositivo que sirva de fuente de vídeo (en principio, HD) que puede ser:

- Ordenador portátil.
- PC de sobremesa
- Smartphone
- Vídeoconsola
- Cámara web
- Etc...

En definitiva, cualquier dispositivo que disponga de una salida HDMI con una resolución de 1280x720 píxeles.

Y finalmente, un dispositivo que sirva para la visualización de la imagen procesada, ya sea:

- Un monitor.
- Un televisor.
- Un proyector.

En definitiva, cualquier dispositivo con una entrada VGA que soporte, al menos, la resolución antes mencionada: 1280x720 píxeles.

Se modelarán dos diseños:

- Un diseño, en el que se recoja el vídeo desde el dispositivo de captura (cámara), se almacene en la memoria de la placa de desarrollo, se realice un procesado por software, y se muestre por pantalla.
- Un segundo diseño, en el que el procesado del vídeo, se realice en un módulo hardware programado para ello.

Se busca también obtener un diseño que, a la vez que potente y eficiente, sea fácilmente modificable y modulable, es decir, que un programador no experto en diseño hardware sea capaz de modificar tanto en la arquitectura, como en el procesado del vídeo en tiempo real.

Es por esto que la arquitectura final cuenta con 3 VDMAs con 3 propósitos diferentes:

1. Un VDMA para controlar la entrada del vídeo.
2. Otro VDMA para controlar la salida del vídeo.
3. Un último VDMA para controlar el core de procesamiento del vídeo.

## 1.4 Posibles aplicaciones y diseños similares

### 1.4.1 Separar materiales en una cadena de montaje

Uno de los sistemas para los que es muy útil un equipo de detección de bordes, puede ser una cadena de montaje en la que tres brazos robóticos estén preparados para, respectivamente:

1. Coger un tornillo y colocarlo en su sitio.
2. Coger una arandela y ponerla en el tornillo
3. Coger una tuerca y ajustarla al tornillo anterior para fijar una pieza.

Una tarea que en principio parece sencilla, puede complicarse cuando tanto tornillos, tuercas y arandelas se reparten en la misma cinta transportadora, haciendo necesario que los brazos robóticos tengan que diferenciar entre tornillo y tuerca para coger lo que tengan programado.

En un principio, “los ojos” del brazo robótico, verían algo similar a esto:



Ilustración 2. Ejemplo visión tornillos/tuercas

Una vez que se realiza el filtrado de la imagen, y se aplica la detección de bordes por Sobel, se obtiene una visión como esta:



Ilustración 3. Imagen tratada con detección de bordes

Tras el proceso de detección de bordes, se consigue una imagen más fácil de tratar, pues aplicando un valor umbral, puede conseguirse una imagen con valores binarios. Indicando con 0 las partes negras, y con 1 las partes blancas, mejorando aún más la imagen detectada por el brazo robótico y por tanto simplificando el algoritmo posterior para identificación de los objetos.

De esta forma el brazo robótico estaría consiguiendo el procesado de la imagen en tiempo real realizado por el sistema de procesamiento de visión. El proceso de selección del material de la cinta transportadora, sólo llevaría un tiempo establecido por el movimiento del brazo, pues todo el procesado lo realizaría el hardware (en nuestro caso, la Zybo Zynq) dejando al brazo robótico únicamente la tarea de situarse en las coordenadas que le indique el software encargado para ello.

#### 1.4.2 Identificación de matrículas

Otra gran aplicación que puede dársele a la detección de bordes es la de identificar una matrícula de un vehículo, que podría ser un coche, una moto, o cualquier tipo de vehículo en función de la complejidad que se añada al procesado.

Estos sistemas se utilizan en gran número de parking públicos en los que, para dejar el vehículo, se hace un control de la matrícula del mismo, con el objetivo de tener más seguridad con los vehículos que entran y salen, así como de evitar estafas por

intentar utilizar el ticket de un vehículo, para salir con otro o sacar un nuevo ticket sin vehículo para pagar menos por estacionar.

Al utilizar un sistema que controla la matrícula del vehículo, identificador único del mismo (al menos, en teoría), se consigue evitar pérdidas para el aparcamiento y además añade un nivel de seguridad pues se puede consultar una base de datos creada con los vehículos que han entrado y salido, así como de las horas de los acontecimientos.

A través de la detección de bordes se puede obtener la posición aproximada de cada uno de los números de la matrícula y posteriormente se haría un recorte individual de éstos, seguido de un análisis para reconocer la numeración y las letras de la matrícula.

## 1.5 Materiales y métodos.

Para el desarrollo de todo este proyecto fin de carrera, se han utilizado los siguientes dispositivos:

### 1.5.1 Materiales

#### 1.5.1.1 Ordenador portátil.

Se ha utilizado a la hora del desarrollo tanto para realizar la programación software/hardware, como para realizar las distintas pruebas de funcionamiento y rendimiento. Este portátil cuenta con una salida HDMI que se utiliza como fuente de vídeo a procesar por el diseño realizado. Equipo utilizado: ASUS K55VM. Prestaciones:

- Intel core i7-3610QM (3<sup>a</sup> generación) a 2.30GHz.
- 6 GB memoria RAM DDR3.
- Sistema operativo: Windows 7 Ultimate, 64 bits.
- Tarjeta gráfica integrada: Intel HD 4400
- Tarjeta gráfica dedicada: NVIDIA GeForce GT 630M, 2GB.
- 2 x puertos USB 3.0 + 1 x puerto USB 2.0.
- Salida vídeo HDMI + VGA.

#### 1.5.1.2 Placa de desarrollo Zybo Zynq – 7000.

[DIG ,15, A] Esta placa de desarrollo cuenta con un procesador Cortex A9 de doble núcleo con una frecuencia de reloj de 650 MHz de una memoria RAM de 512 MB DDR3 con 8 canales DMA y una memoria FLASH de 128 Mb.

Además, cuenta con gran cantidad de puertos para conectar periféricos como:

- Puerto JTAG por microUSB para programación y comunicación por terminal.
- Puerto USB 2.0.
- USB OTG.
- Puerto Ethernet 1GB.
- Control de periféricos por I<sup>2</sup>C.
- Puerto HDMI para entrada/salida.

- Puerto VGA de salida.
- Entradas/salidas Jack para audio
- Periféricos de tipo GPIO para lectura de estado de los mismos, 4 Switches, 4 pulsadores y 4 LEDs.
- Varios puertos de expansión.

Y lo más importante, una FPGA reprogramable equivalente a una Artix-7 que cuenta con:

- 28000 celdas lógicas.
- Un bloque de memoria RAM de 240 KB.
- 80 DSPs.
- Chip de doble canal integrado, de 12 bits y un conversor analógico digital.



Ilustración 4. <sup>[SHO,15]</sup> Fotografía de la placa de desarrollo Zybo Zynq 7000

#### 1.5.1.3 Monitor VGA.

Un monitor normal con un puerto de entrada VGA con el único requisito de que debe soportar la resolución con la que se está trabajando: mínimo 1280x720 píxeles.

#### 1.5.2 Métodos

Para la programación, tanto del software como del hardware, se han utilizado herramientas proporcionadas por el fabricante de la placa de desarrollo: Xilinx Inc. Aunque el código de programación es muy similar al código C o C++, se recomienda utilizar las propias herramientas de Xilinx con el objetivo de facilitar el funcionamiento del sistema con la Zybo.

##### 1.5.2.1 Vivado.

Vivado se trata de una herramienta de Xilinx con la que se realiza el diseño de la arquitectura.



Ilustración 5. [DIG, 15, B] Logo Vivado

Es una herramienta que facilita la programación de la misma, utilizando para ello una programación gráfica a base de bloques, denominados “IP Cores” (a partir de ahora, “cores”) que, pueden configurarse de una manera cómoda y sencilla realizando un clic doble sobre ellos. Para todo el desarrollo de este proyecto se ha utilizado la versión de Vivado 2014.4.

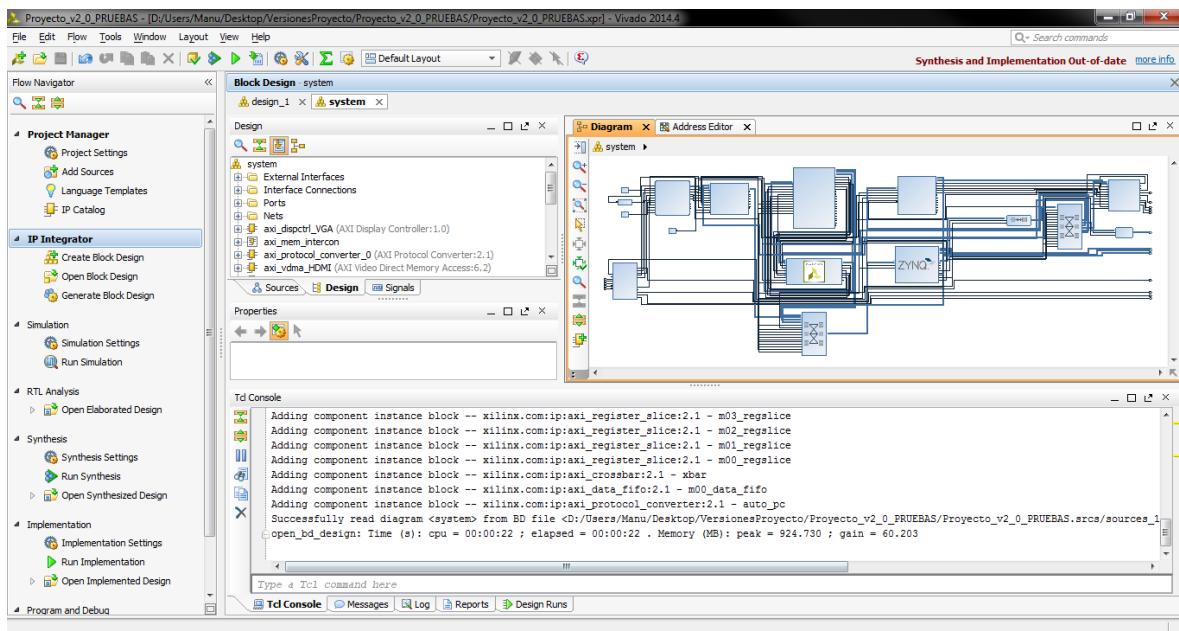


Ilustración 6. Interfaz gráfica de Vivado: Diseño de bloques a la derecha

Además, ofrece algunos menús de conexión y creación del diseño, es decir, nada más crear un nuevo diseño, Vivado ofrece la opción de añadir cores al diseño vacío, y a su vez conforme se van creando estos cores, ofrece la posibilidad de realizar una conexión automática:

## Procesado de vídeo con Zybo Zynq

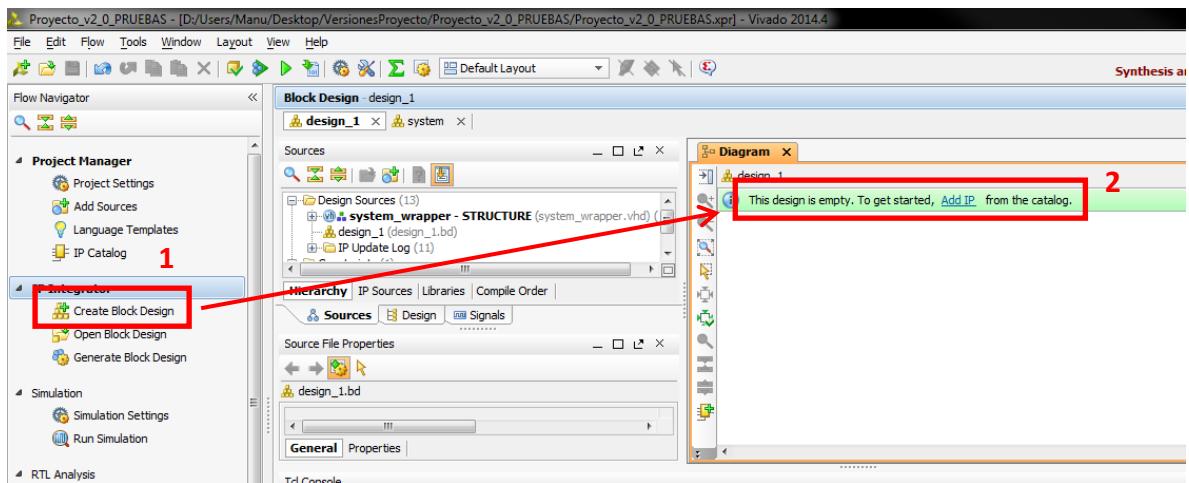


Ilustración 7. Indicación de insertar IP Core en diseño vacío

Y de igual manera, cuando se ha añadido algún core indica, si es posible, que puede realizar una conexión automática. Este caso sólo se da cuando son cores propios de Xilinx, para los cuales Vivado ofrece una ayuda para su conexión:

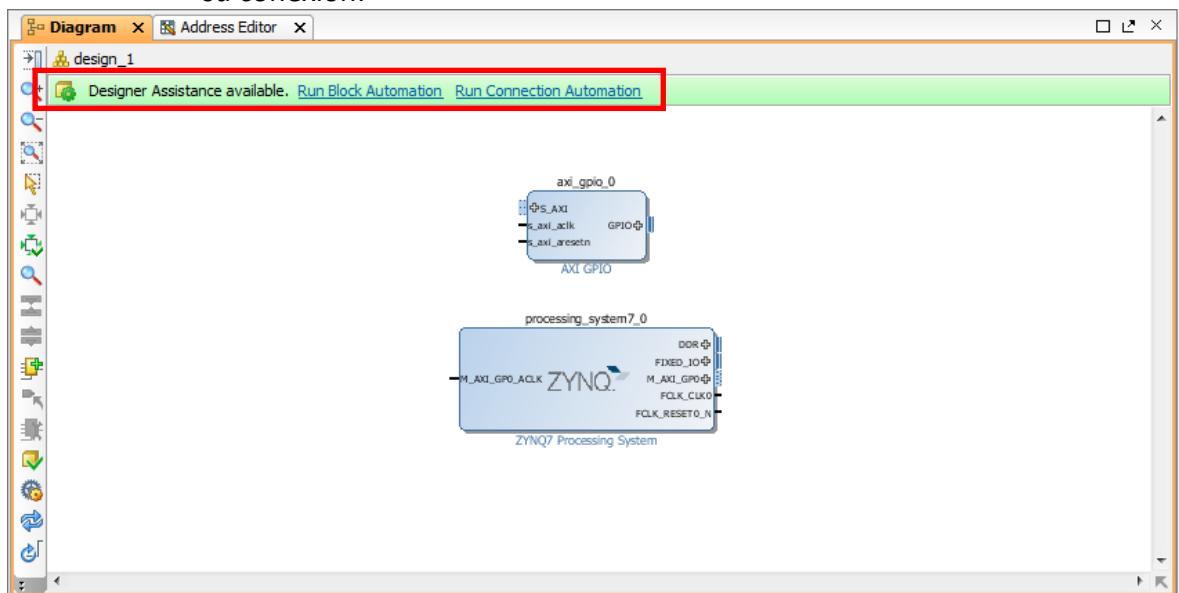


Ilustración 8. Ejemplo de conexión automática 1

Y una vez ejecutada la conexión automática se pueden realizar pequeñas configuraciones:

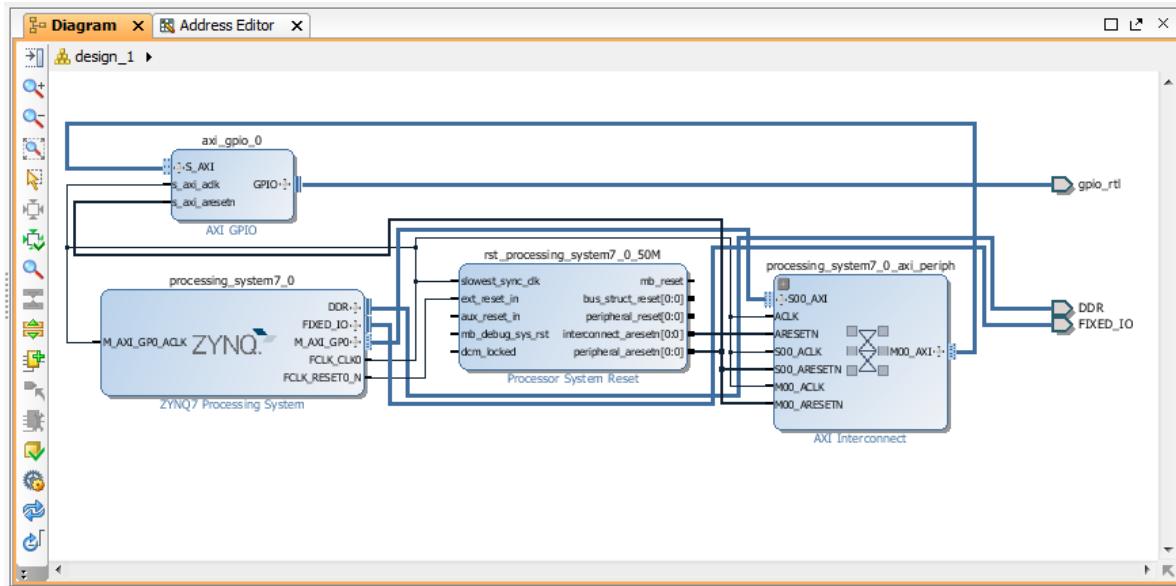


Ilustración 9. Ejemplo de conexión automática 2

Donde, como se puede observar, Vivado ha creado algunos cores necesarios intermedios para las conexiones de una forma automática, facilitando al desarrollador gran parte de su tarea.

Vivado también permite añadir al diseño, líneas de depuración o debug de manera que se pueda visualizar los datos que circulan a través de ellas. Así se facilita la tarea de búsqueda de posibles fallos, mostrando como si de un osciloscopio digital se tratase, el valor de los bits de las líneas marcadas como debug:

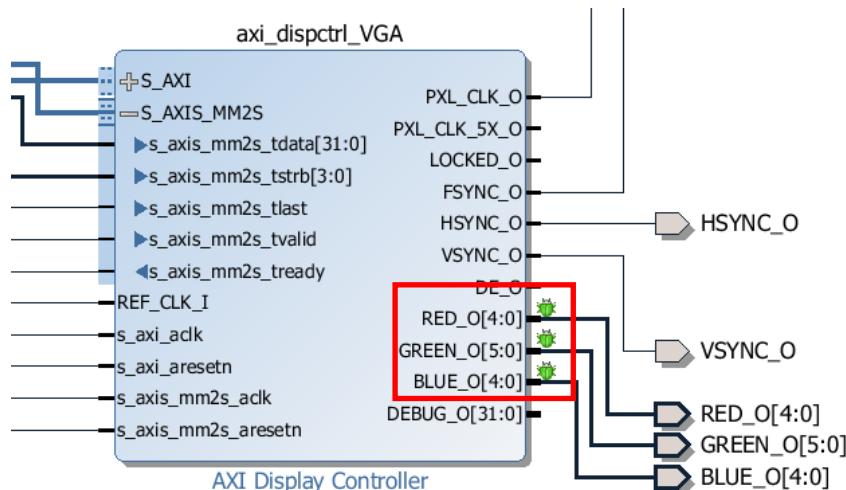


Ilustración 10. Ejemplo de líneas marcadas para debug

La visualización se realiza a través de una herramienta denominada ChipScope, para el cual es necesaria una licencia de uso. Una vez marcadas las líneas para

debug, hay que realizar la síntesis del diseño y configurar el debug, indicando cuántas tomas se quieren visualizar de las señales:

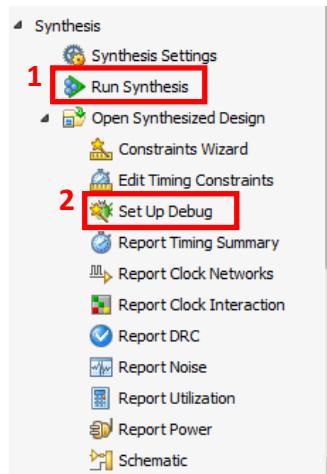


Ilustración 11. Pasos para configurar Debug

Y una vez configurado todo, debe ejecutarse el programa en la placa y después, abrir dentro de Vivado la sección “Program and Debug”.

La vista de las señales en tiempo real, será algo parecido a:

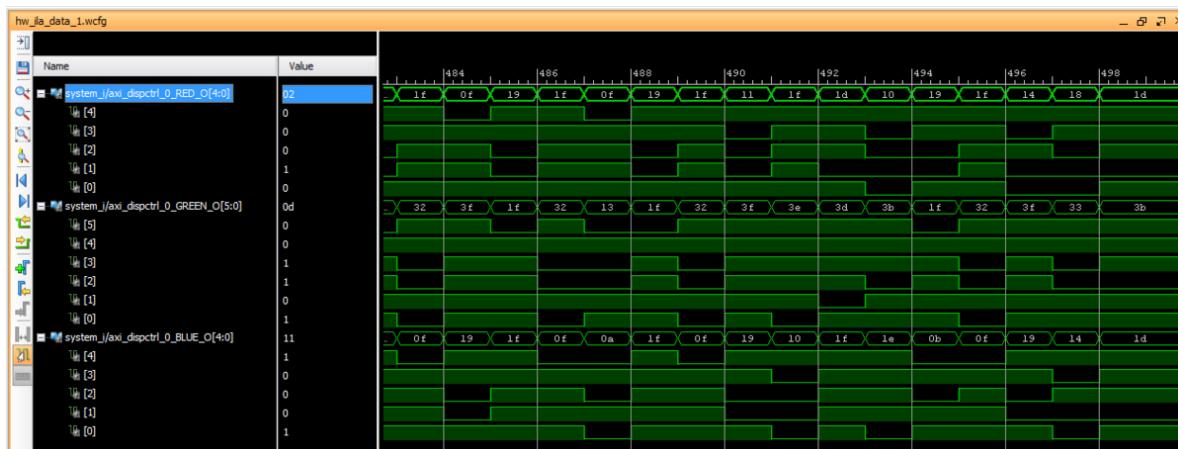


Ilustración 12. Ejemplo ChipScope

Una vez que todo el diseño está realizado, debe sintetizarse, implementarse y generar el bitstream. El bitstream, es un fichero que Vivado genera con el cual la placa se reconfigura, haciendo que la FPGA adapte sus puertas lógicas para que funcione como indique el diseño.

Todo el diseño realizado con los distintos cores en Vivado (Ilustración 6) es el hardware que formará la arquitectura. Este hardware se controla mediante software programado en Xilinx SDK utilizando para ello los distintos drivers de funcionamiento de los cores y el código que el programador quiera utilizar.

El bitstream debe exportarse junto con el hardware, para poder después utilizarlo con Xilinx SDK y ejecutarse en la placa de desarrollo con el fin de poder visualizar todo el desarrollo.

### 1.5.2.2 Xilinx SDK

Esta herramienta de Xilinx es un entorno de desarrollo basado en Eclipse, con una interfaz muy parecida pero adaptada a los productos de Xilinx. Para todo el desarrollo de este proyecto se ha utilizado la versión 2014.4 de Xilinx SDK.

Una vez que en Vivado, se ha exportado el hardware (que incluye toda la arquitectura de bloques junto con su configuración) y el bitstream (para que la FPGA se reconfigure como deba) debe lanzarse el Xilinx SDK (De aquí en adelante SDK) desde el mismo menú de Vivado:

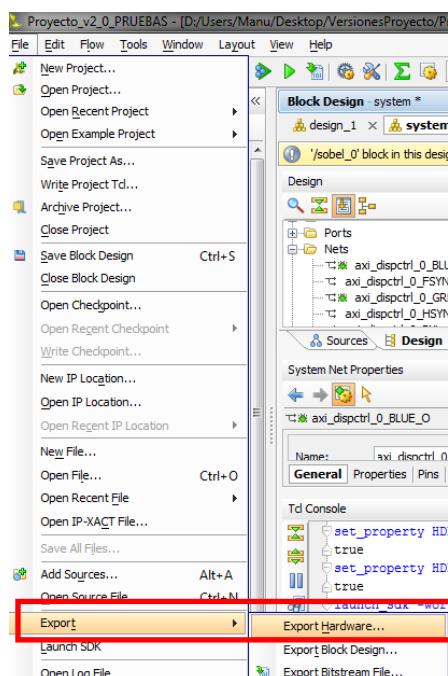


Ilustración 13. Exportar hardware (1)

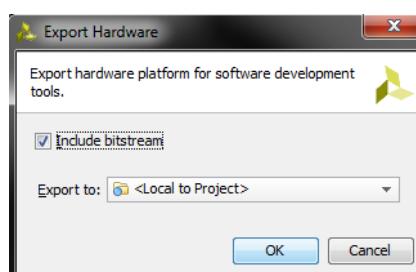


Ilustración 14. Exportar hardware (2)

## Procesado de vídeo con Zybo Zynq

Y una vez que tenemos el hardware exportado, lanzamos el SDK:

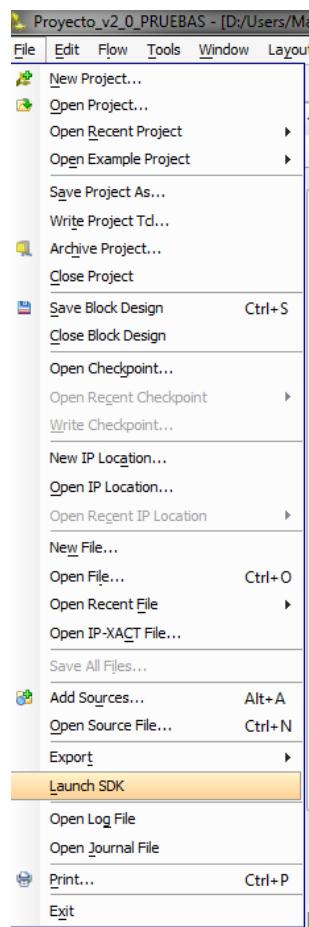


Ilustración 15. Lanzar el SDK desde Vivado

La interfaz del SDK es la siguiente:

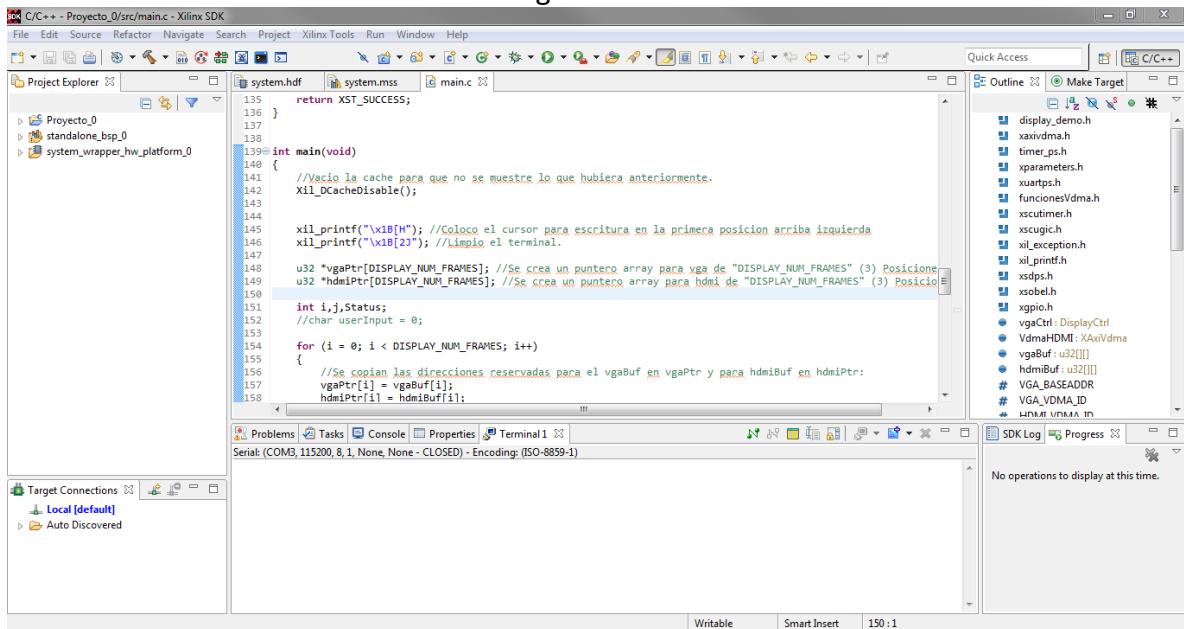


Ilustración 16. Interfaz gráfica de Xilinx SDK

El propio SDK, se encarga de crear un hardware platform, que incluye todos los cores insertados en el diseño desde Vivado y aporta información sobre las direcciones de memoria donde se han situado. Esto se realiza de forma automática.

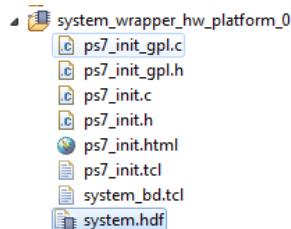


Ilustración 17. Hardware platform creado por Xilinx SDK

A continuación, debe crearse un BSP, que incluye todos los códigos necesarios para el manejo de estos cores, ya sean propios de Xilinx, o cores HLS como el generado para este diseño, encargado de realizar el procesado del vídeo. Estos códigos, son las librerías o drivers que aportan información sobre cómo deben iniciarse y configurarse los cores hardware dentro del software.

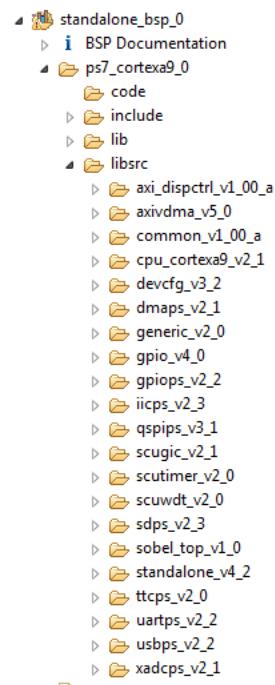


Ilustración 18. BSP con todas las librerías que incluye

Finalmente se crea un Application Project, el cual será el proyecto donde generaremos el código para nuestra aplicación que será el que debe ejecutarse en la placa de desarrollo para poder ver el resultado final.

El programa se realiza con lenguaje de programación C o C++ (C en mi caso) que luego, el SDK es capaz de interpretar y ajustar a la placa de desarrollo.

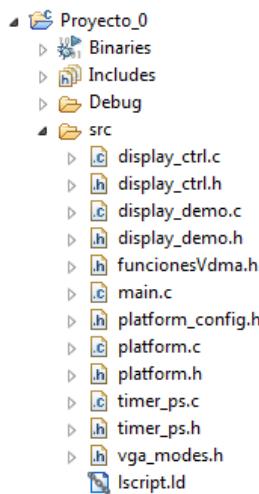


Ilustración 19. Proyecto con todos los ficheros de código C

Para terminar, una vez que todo el proyecto está desarrollado, se debe cargar el bitstream en la placa de manera que se reprograme la FPGA con la arquitectura diseñada. Para ello, se conectará la Zybo Zynq mediante un cable microUSB al PC:

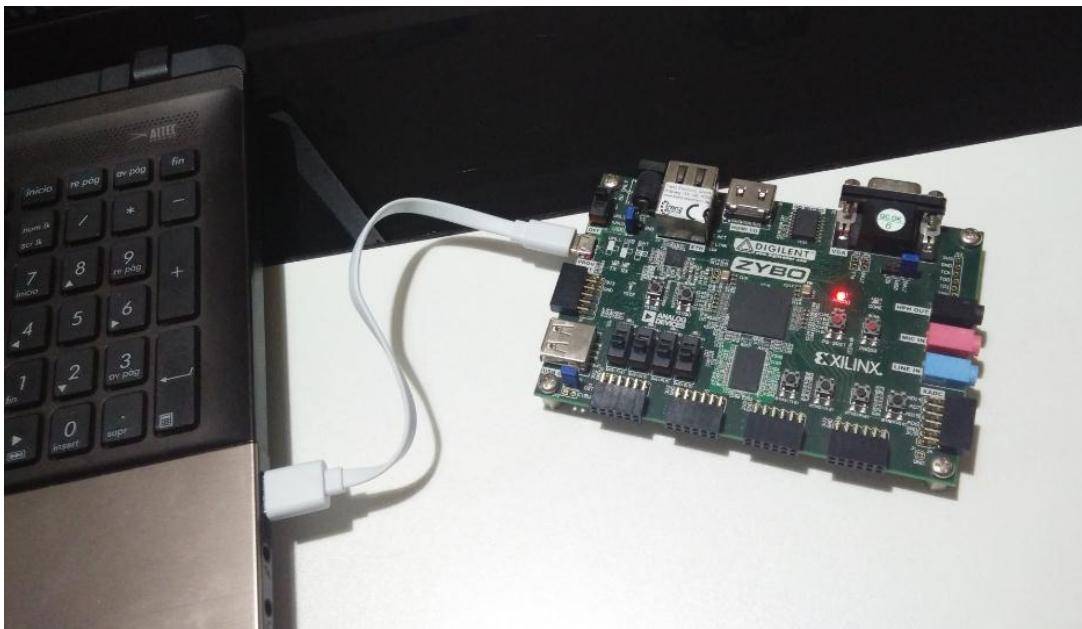


Ilustración 20. Conexión Zybo - PC mediante cable microUSB

El LED rojo indica que la placa está alimentada y encendida. Tras esta conexión se debe hacer la descarga del bitstream a la placa, pulsando el botón:

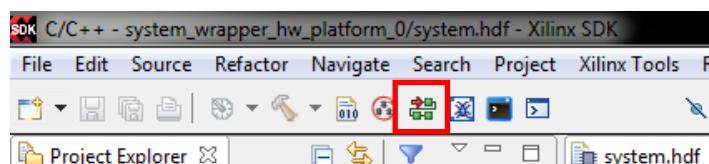


Ilustración 21. Botón de descarga del bitstream

Cuando el proceso de descarga se ha realizado correctamente, la Zybo lo indica iluminando el LED LD10 “DONE” en verde:

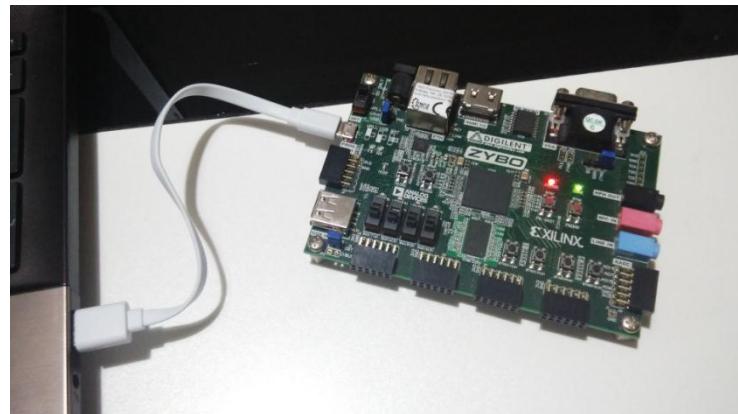


Ilustración 22. Placa con bitstream cargado

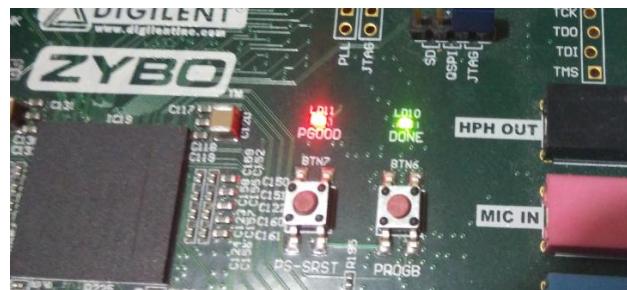


Ilustración 23. Detalle del LED LD10 "DONE" iluminado

Para finalizar, se debe ejecutar el proyecto realizado en la placa, eligiendo para ello la opción “Run as → Launch on hardware” del menú contextual del proyecto:

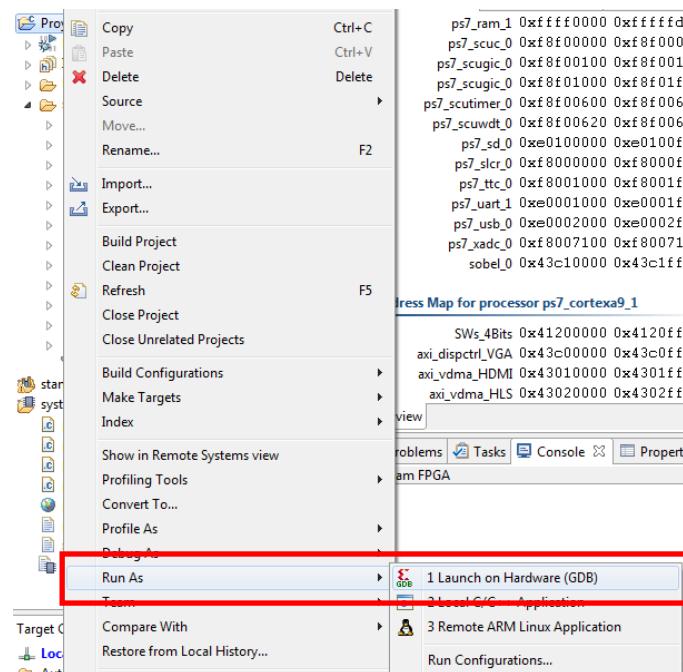


Ilustración 24. Ejecutar proyecto en la placa de desarrollo

### 1.5.2.3 Vivado HLS

[XIL,15,C],[LOU,ROS,MAR,DAV,15] Esta herramienta de Xilinx, permite al desarrollador generar IP cores de una forma sencilla y rápida, con bajos conocimientos de ingeniería hardware, pero a su vez consiguiendo unos diseños eficientes y potentes.



Ilustración 25. Logo Vivado HLS

Para ello, Vivado HLS permite programar los cores con un lenguaje tipo C o C++ pero con algunas mejoras. Este lenguaje permite añadir directivas al código haciendo que el core que se está programando, se centre más en conseguir eficiencia al cumplir tiempos de ejecución, o bien, a la hora de ahorrar espacio.

Esta herramienta, debe instalarse por separado a Vivado y Xilinx SDK. La versión utilizada durante todo el desarrollo de este proyecto, ha sido la versión 2014.4 de Vivado HLS.

La interfaz gráfica de Vivado HLS, es muy similar a la ya presentada de Xilinx SDK, con ciertas diferencias.

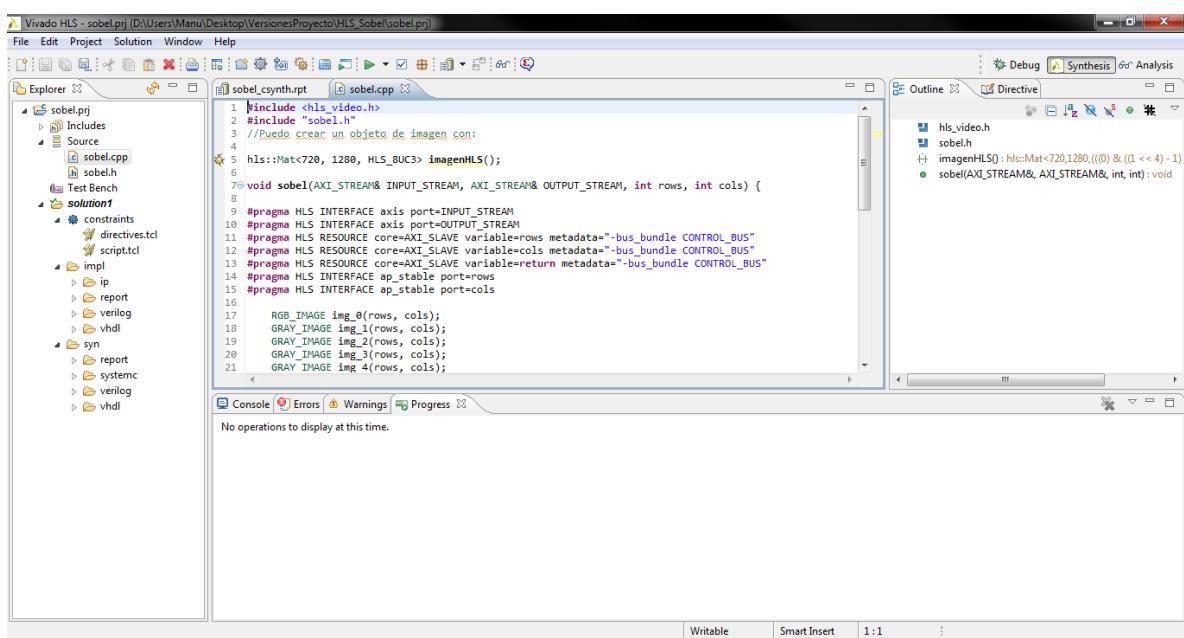


Ilustración 26. Interfaz gráfica de Vivado HLS

En el explorador de ficheros, nos encontramos únicamente con el proyecto que se esté trabajando, sin aludir al hardware diseñado. Además nos encontramos con un nuevo apartado en la interfaz a la derecha, donde lo más importante es la pestaña de “Directive”

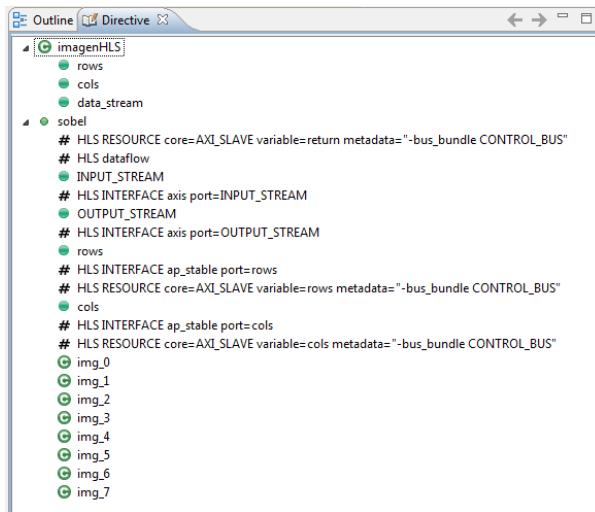


Ilustración 27. Detalle de pestaña "Directive"

Es aquí donde se pueden definir las diferentes directivas a incluir en el proyecto, para conseguir la mejor solución que se requiera. En la imagen anterior se puede ver cómo se han definido que las variables return, rows y cols estén diseñadas como un recurso, que estará disponible a través de un bus el cual se ha denominado “CONTROL\_BUS”.

A su vez, los streams de entrada y salida, se han definido como interfaces de tipo AXI Stream (axis). Todas estas directivas, pueden añadirse pulsando el botón secundario sobre las variables y utilizando el menú que nos proporciona la herramienta.

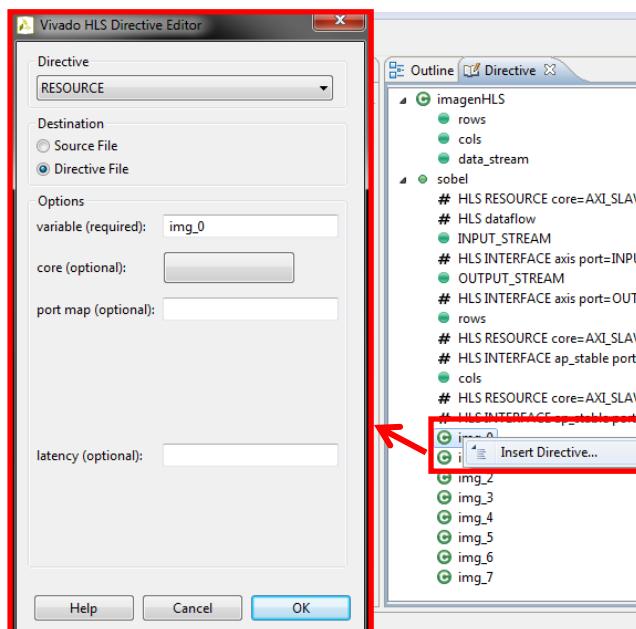


Ilustración 28. Detalle menú "Insert directive..."

Una vez se realiza la programación, el propio Vivado HLS establecerá cuáles de las señales serán entradas y cuáles salidas de manera que el programador no tiene que preocuparse por establecerlas pues en función de su uso en el código se les asignará la dirección apropiada.

Además, Vivado HLS informa al desarrollador de la dirección que ha establecido para cada una de las interfaces que se hayan establecido a través de las directivas:

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
INPUT_STREAM_TDATA	in	32	axis	INPUT_STREAM_V_data_V	pointer
INPUT_STREAM_TKEEP	in	4	axis	INPUT_STREAM_V_keep_V	pointer
INPUT_STREAM_TSTRB	in	4	axis	INPUT_STREAM_V_strb_V	pointer
INPUT_STREAM_TUSER	in	1	axis	INPUT_STREAM_V_user_V	pointer
INPUT_STREAM_TLAST	in	1	axis	INPUT_STREAM_V_last_V	pointer
INPUT_STREAM_TID	in	1	axis	INPUT_STREAM_V_id_V	pointer
INPUT_STREAM_TDEST	in	1	axis	INPUT_STREAM_V_dest_V	pointer
INPUT_STREAM_TVALID	in	1	axis	INPUT_STREAM_V_dest_V	pointer
INPUT_STREAM_TREADY	out	1	axis	INPUT_STREAM_V_dest_V	pointer
OUTPUT_STREAM_TDATA	out	32	axis	OUTPUT_STREAM_V_data_V	pointer
OUTPUT_STREAM_TKEEP	out	4	axis	OUTPUT_STREAM_V_keep_V	pointer
OUTPUT_STREAM_TSTRB	out	4	axis	OUTPUT_STREAM_V_strb_V	pointer
OUTPUT_STREAM_TUSER	out	1	axis	OUTPUT_STREAM_V_user_V	pointer
OUTPUT_STREAM_TLAST	out	1	axis	OUTPUT_STREAM_V_last_V	pointer
OUTPUT_STREAM_TID	out	1	axis	OUTPUT_STREAM_V_id_V	pointer
OUTPUT_STREAM_TDEST	out	1	axis	OUTPUT_STREAM_V_dest_V	pointer
OUTPUT_STREAM_TVALID	out	1	axis	OUTPUT_STREAM_V_dest_V	pointer
OUTPUT_STREAM_TREADY	in	1	axis	OUTPUT_STREAM_V_dest_V	pointer
rows	in	32	ap_stable	rows	scalar
cols	in	32	ap_stable	cols	scalar
ap_clk	in	1	ap_ctrl_hs	sobel	return value
ap_rst_n	in	1	ap_ctrl_hs	sobel	return value
ap_start	in	1	ap_ctrl_hs	sobel	return value
ap_done	out	1	ap_ctrl_hs	sobel	return value
ap_idle	out	1	ap_ctrl_hs	sobel	return value
ap_ready	out	1	ap_ctrl_hs	sobel	return value

Ilustración 29. Ejemplo de informe de señales Vivado HLS

El punto más importante por el que se debe mencionar Vivado HLS es por el core que genera una vez que se ha hecho toda la síntesis del código del proyecto.

Vivado HLS crea un fichero comprimido en formato “.zip” que se puede importar en Vivado de manera que pueda incluirse en el diseño de bloques y conectarlo al resto del sistema para que realice su función.

Con todo esto, se consigue crear el hardware de una forma sencilla, mediante programación en alto nivel, pero como ya se ha mencionado, grandes opciones de optimización y flexibilidad. Una vez que el core creado por Vivado HLS esté integrado, se podrá modificar su código y posteriormente desde Vivado, actualizar el core de una forma sencilla y automatizada por Vivado:

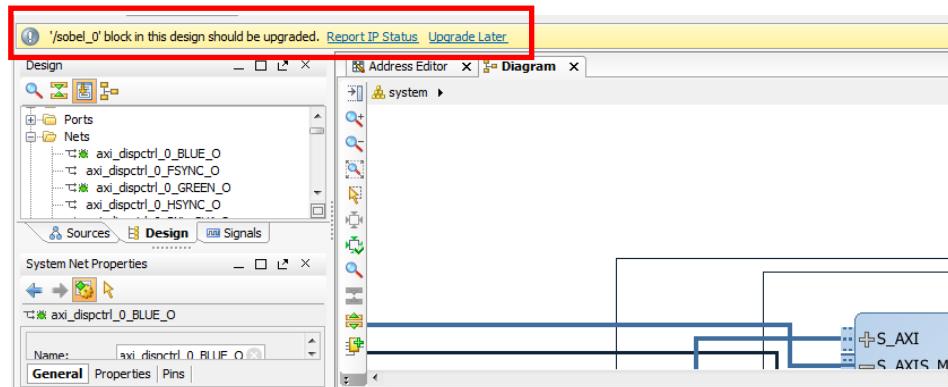
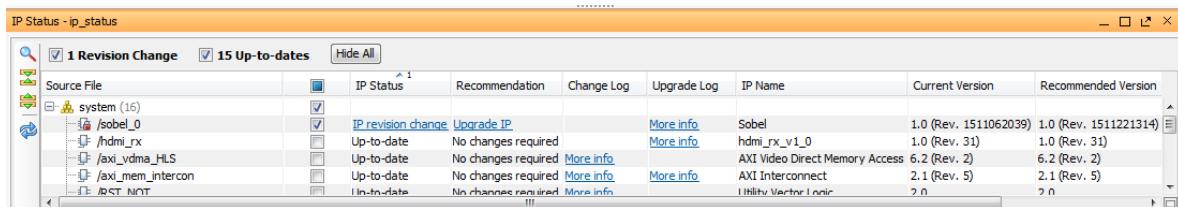


Ilustración 30. Aviso de Vivado de que hay un core sin actualizar

Donde, pulsando sobre “Report IP Status” obtendremos un listado de todos los cores incluidos en el diseño y cuáles necesitan actualizarse porque se ha encontrado una versión posterior en los repositorios, con información detallada de los mismos:



The screenshot shows the 'IP Status - ip\_status' window in Vivado. The table lists various IP components under the 'system' source file, categorized by their status: 'Up-to-date' or 'IP revision change'. For each component, it shows its name, current version, and recommended version. For example, 'Sobel' is at version 1.0 (Rev. 1511062039) and recommended at 1.0 (Rev. 1511221314). Other components like 'hdmi\_rx\_v1\_0', 'AXI Video Direct Memory Access', and 'AXI Interconnect' also have their respective versions and recommended updates listed.

Ilustración 31. Ejemplo del "Report IP Status" de Vivado

Permitiendo realizar la actualización del mismo sin tener que modificar de ninguna manera el diseño y consiguiendo que el sistema esté actualizado, realizando de nuevo la generación del bitstream.

## 1.6 Marco de trabajo.

Este proyecto podría incluirse dentro de muchos marcos de trabajo, pues el fin de realizar captura de vídeo con procesado del mismo en tiempo real puede aplicarse a muchas aplicaciones distintas y de distintos ámbitos. Sin embargo, algunos proyectos donde puede encajar muy bien este desarrollo pueden ser:

### 1.6.1 Proyecto VITVIR

<sup>[ENR, 15]</sup>VITVIR, se trata de un proyecto de Excelencia de la Junta de Andalucía (TIC-8120) cuya misión es el desarrollo de un sistema de visión, capaz de extraer la información en tres dimensiones (3D) de la escena que se visualiza, a través de información obtenida con varias cámaras o sensores 3D como pueden ser cámaras TOF.

En VITVIR: Visión TriDimensional para Vídeoanálisis Interactivo y Realidad Aumentada, los objetos de la escena se clasifican entre:

- **Estáticos:** Muros, columnas, muebles, etc...
- **Dinámicos:** Mobiliario móvil, personas, mochilas, etc...

Como elementos base para determinación de acciones a realizar en diferentes entornos de aplicación como:

- Vídeovigilancia
- Realidad aumentada
- Domótica.
- Etc.

Éste proyecto podría utilizarse dentro de VITVIR a la hora de realizar la captura del vídeo, pues con la arquitectura diseñada podría realizarse el tratamiento del vídeo dentro del core ya realizado, simplemente aplicando algunos ajustes a nivel de software.

En cuanto a la captura del vídeo, sería necesario modificar en la arquitectura únicamente aquellos cores que se usan para captura y muestra del vídeo, pues como ya se ha comentado, se ha conseguido un sistema fácilmente modulable.

Es por esto, por lo que es importante destacar el aspecto modulable del diseño, pues puede adaptarse de una forma sencilla a distintos enfoques y periféricos.

### 1.6.2 Proyecto MULTIVISION

[ENR, 15] Este proyecto también es un Proyecto de Excelencia de la Junta de Andalucía, en la que se presenta un sistema de visión en tiempo real multi-cámara para interpretación de escenas (TIC-3873).

Se aborda el desarrollo de un sistema de visión híbrido hardware/software, en tiempo real usando para ello múltiples cámaras. En todo sistema de visión, el objetivo es traducir imágenes a una información concreta, es decir “Datos extraídos de la ‘interpretación de la escena’”. Es por eso, que en el proyecto MULTIVISION, se estudian distintos esquemas de visión que permitan el procesamiento eficiente de las imágenes extraídas de varias cámaras y se realiza un tratamiento de forma complementaria de las estimaciones de las distintas cámaras para realizar la “Interpretación de la escena”.

Un escenario barrido por múltiples cámaras, genera una gran cantidad de datos que se debe procesar en tiempo real, es por ello que se utilizan algoritmos de procesamiento de la imagen a bajo nivel, como el color, el movimiento... Separando cada tarea y distribuyéndolas en diversos dispositivos FPGA como núcleos de procesamiento, mientras que tareas de más alto nivel, se realicen sólo de forma global y centralizada.

Por estas razones, el diseño propuesto en este proyecto basado en una placa Zybo Zynq, la cual incluye una potente FPGA y cuya arquitectura aquí presentada es muy modular y sencilla para el desarrollador. Podría también adaptarse al uso que se necesite en MULTIVISION, pues el procesado de la imagen puede realizarse utilizando Vivado HLS como herramienta de desarrollo encargada de traducir el lenguaje de alto nivel (C, C++...) a un lenguaje de bajo nivel, capaz de conseguir la mayor optimización.

Además se pueden incluir distintas directivas para la generación del código a bajo nivel para optimizar aquello que sea vital, como el espacio en placa, el tiempo de procesado, la segmentación, etc...

Es interesante que, con un par de cores diferentes, pueda cambiarse la fuente de entrada de vídeo y la salida del mismo, sin tener que modificar nada más de la arquitectura.

Otro añadido es el hecho de que la placa de desarrollo Zybo Zynq que se utiliza en esta arquitectura, es un sistema sencillo y barato, pero a la vez lo suficientemente potente para esta tarea, por lo que se reducirían costes a la hora de aplicarlo sobre varias cámaras.

## 1.7 Estructura de la memoria.

El siguiente proyecto, está dividido en 6 secciones:

La primera sección, es la introducción al proyecto y la motivación del mismo. También se realiza una pequeña introducción de los sistemas empotrados y se explica algo más en profundidad los materiales y métodos. Además se ha descrito el marco de trabajo, referenciando algunos proyecto donde éste podría encajar y aludiendo a distintos ejemplos donde puede aplicarse este desarrollo.

El segundo capítulo se comentan algunos sistemas empotrados que existen en la actualidad y las características que los hacen importantes, así como de la estrategia de diseño a seguir en la creación de un sistema embebido.

En el tercer capítulo se presenta el diseño realizado, de dónde se parte y por qué, y mostrando varios ejemplos de la realización para que pueda ser útil en el futuro a otras personas que lo requieran.

Se continúa con algunas conclusiones personales tras haber realizado todo este proyecto y se termina con una pequeña bibliografía de todas las fuentes utilizadas para el proyecto y un anexo explicando los distintos cores usados en el diseño.

## 2 Sistemas de visión embotrada.

---

[WIK, 15, D] Los sistemas empotrados o embebidos, son un sistema de computación diseñado para realizar una o algunas tareas dedicadas, es decir, tareas particulares, frecuentemente en un sistema de computación en tiempo real y, aunque ya se ha mencionado, no es un sistema como los ordenadores de uso personal, diseñados para cubrir un amplio rango de necesidades sino que se diseñan para cubrir necesidades específicas.

Al contrario de lo que mucha gente podría pensar, los sistemas embebidos se encuentran en muchos de los lugares que diariamente transitamos, pero el desconocimiento de estos, hace que el usuario de a pie no sepa identificarlos.

Dispositivos como:

- Taxímetros.
- El sistema de control de falsificación de billetes de una máquina de los supermercados
- Los móviles inteligentes que hoy en día prácticamente todo el mundo lleva en el bolsillo.
- Automóviles, que pueden contener hasta 40 sistemas embebidos distintos, como el sistema de elevalunas, cierre centralizado, del airbag, el ESP, el ABS...
- Periféricos de juego, como mandos, joysticks, cámaras...
- En electrodomésticos de todo tipo: Microondas, lavavajillas...
- Y un larguísimo etcétera...

Podríamos resumir por tanto, que un sistema embebido es un “ordenador” especializado para una solución en particular, en la que normalmente es necesario:

1. Usar una solución óptima para la tarea o tareas a resolver.
2. Normalmente es una parte “especializada” que se instala en algún sistema anfitrión (como pasa en los coches).
3. Se dota únicamente de aquellos módulos que sean exclusivamente necesarios para llevar a cabo su tarea, consiguiendo con esto una optimización en el espacio y una reducción de los costes de fabricación del sistema.

[SYN,15] El hecho de utilizar un sistema embebido, ofrece un gran añadido a un producto, haciendo que claramente se distinga de productos de la competencia, ya sea por cómo están diseñados sus componentes, cómo está programado, su forma e integración en el sistema anfitrión... Todos los sistemas empotrados, en general, pertenecen a máquinas que se espera que funcionen durante muchos años, programándose incluso de forma que sean capaces de recuperarse de algún problema que pudiera surgir de una forma autónoma.

Un buen ejemplo de ellos serían los satélites o las sondas espaciales que evidentemente, no son accesibles para poder repararse en caso de una avería. Si lo pensamos, muchos sistemas podrían tener este tipo de problemas, por ejemplo:

- ☒ Estos satélites o sondas, no pueden apagarse y son inaccesibles para repararse.
- ☒ Los sistemas de navegación aérea, sistemas de control de incendios o de control de emisiones de gas, no pueden apagarse por motivos de seguridad.
- ☒ Pueden ocasionar pérdidas millonarias a la empresa que las utilice si el sistema se apaga, como por ejemplo con sistemas de control de empaquetado en fábricas o similares.

A la hora de diseñar un sistema empotrado, deben tenerse en cuenta varias propiedades que debe cumplir, como que por ejemplo que deba ser un sistema robusto y con una muy baja tasa de error (sistemas críticos como un brazo biónico) o bien, un sistema muy reducido por el pequeño espacio donde se va a colocar (un marcapasos) o gran rendimiento utilizando, si fuera necesario, acoplar varios sistemas embebidos para conseguir más potencia ya que no importa el espacio.

[JAV,10, A] Es por esto, que se debe tener en cuenta una estrategia de diseño que se puede estructurar en:

1. Realizar un estudio del flujo de datos.
2. Realizar un estudio de la precisión requerida en cada etapa.
3. Profundidad o tamaño de palabra y si será necesaria aritmética fija o en punto flotante.
4. Estudiar si se utilizará segmentación del cauce de datos.
5. Manejo eficiente de memoria externa y/o bancos de memoria accesibles en paralelo.
6. Replicar circuitos de procesamiento

7. Usar tablas de consulta o bien, cores optimizados para operaciones concretas.
8. Multiplicar el sistema en caso de necesitar más prestaciones.
9. Terminar con un estudio del ancho de banda para habilitar un flujo de datos eficiente.

Con todo esto se conseguiría optimizar el funcionamiento del diseño, en función de las necesidades finales del mismo y consiguiendo un sistema mejor preparado para su tarea.

## 3. Arquitecturas y diseños

---

### 3.1 Diseños de referencia

#### 3.1.2 Zybo Base Design

<sup>[GIT,15]</sup>Para el diseño del sistema, primero se ha partido del sistema proporcionado por Xilinx, adaptado a la versión de desarrollo que se ha utilizado en este proyecto. El diseño ofrecido por el fabricante, “Zybo Base Design” estaba preparado para la versión 2013.4 de VIVADO, por lo que se hizo una portabilidad del sistema a la versión 2014.4.

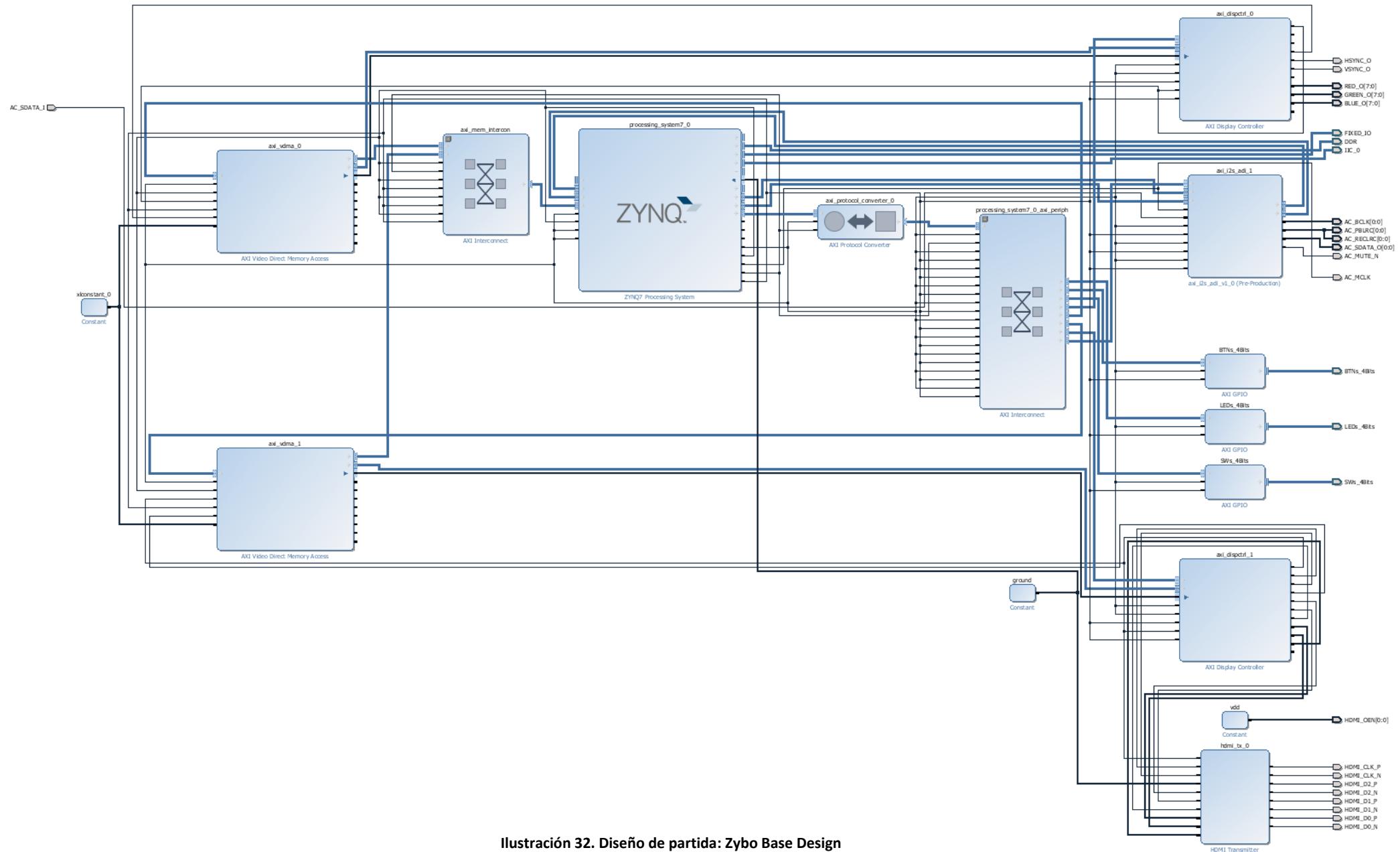


Ilustración 32. Diseño de partida: Zybo Base Design

Este es un diseño de ejemplo del fabricante, en el que se utilizan todos los periféricos de la Zybo, así como todas sus entradas/salidas de modo que el usuario pueda ver que todo funcione correctamente, y obtener un punto de partida para sus diseños.

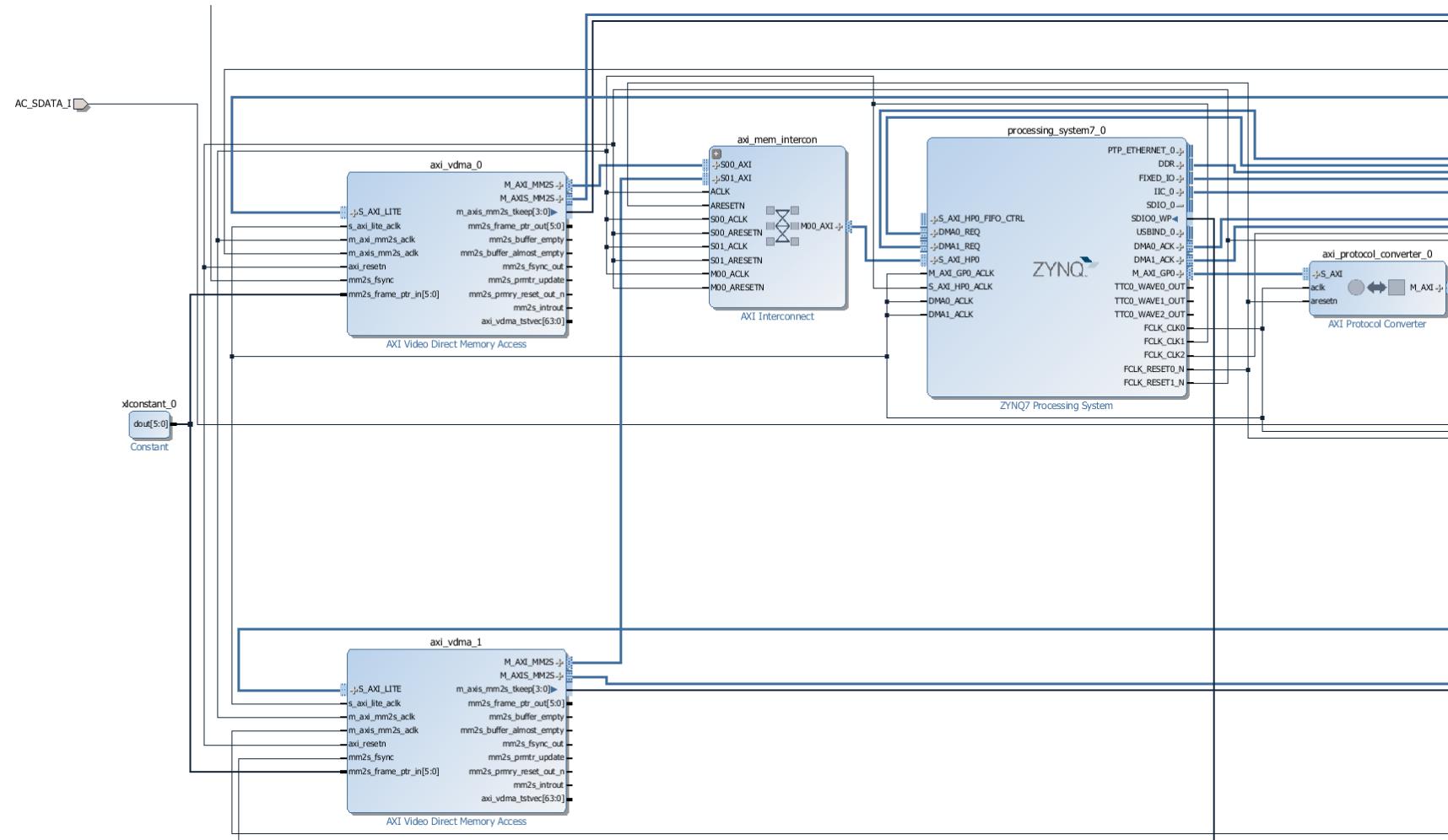


Ilustración 33. Detalle del diseño base (1).

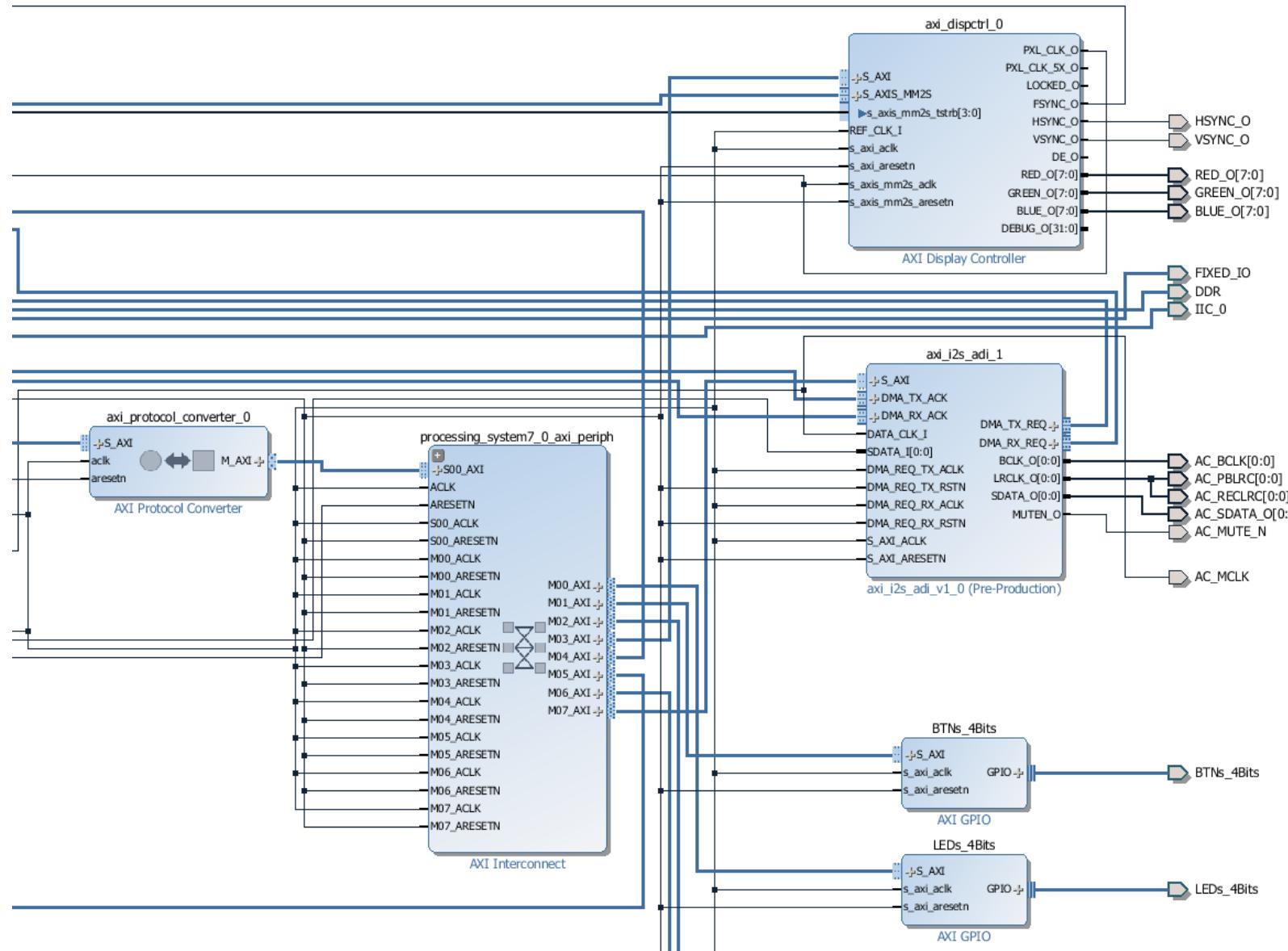


Ilustración 34. Detalle del diseño base (2)

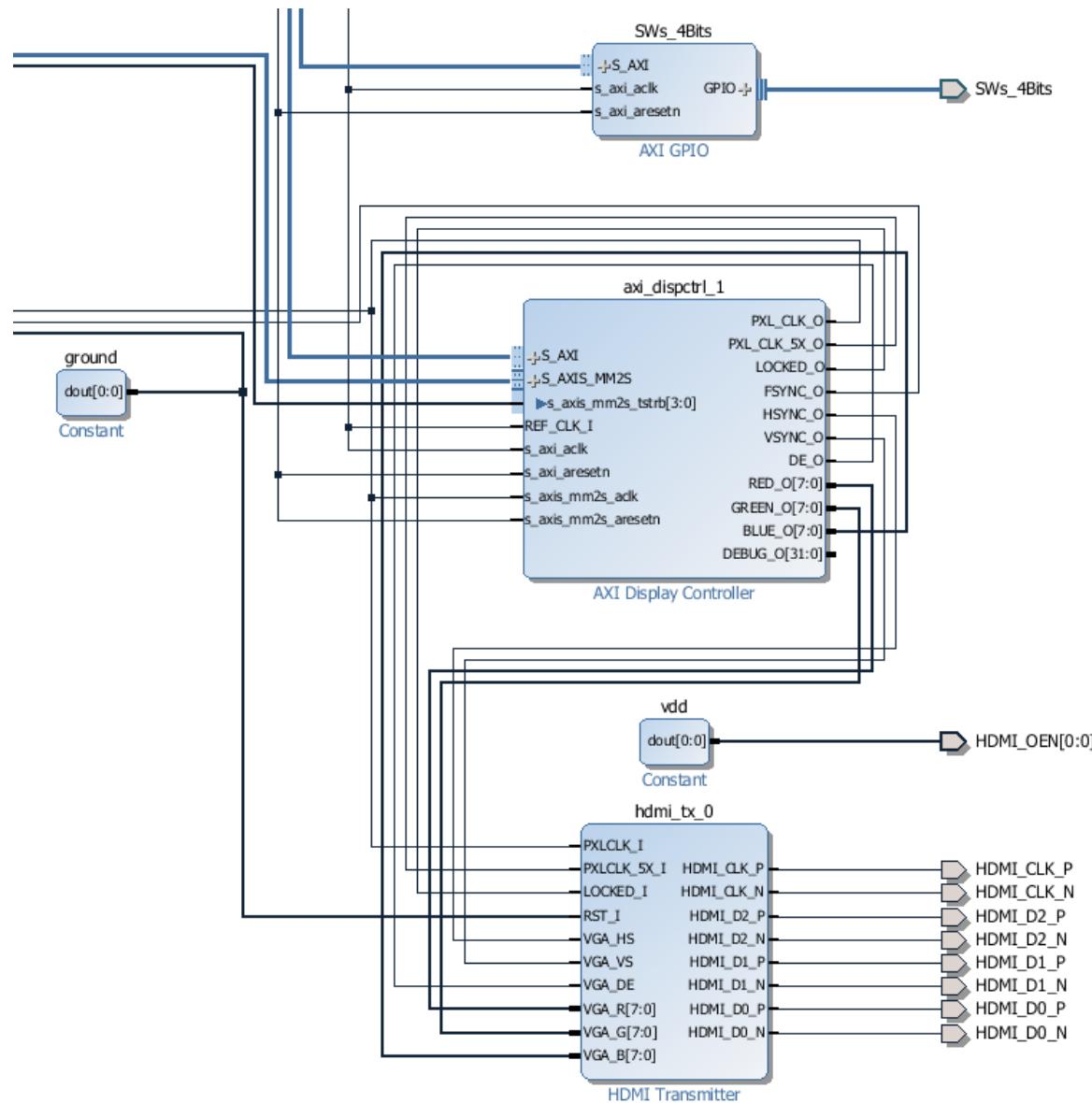


Ilustración 35. Detalle del diseño base (3)

El ejemplo contiene:

- a) Salida de imagen generada por software mostrada tanto por el HDMI como por el VGA.
- b) Un sistema de testeo de los LEDs y botones de la tarjeta, haciendo que, al presionar cada botón, se encienda el LED correspondiente.
- c) Passthrough de audio desde los canales del Jack de ENTRADA al Jack de SALIDA de audio de la tarjeta.
- d) Un sistema de prueba para los switches de la misma, haciendo que se genere un tono más o menos agudo usando los jacks de salida de audio de la Zybo.

Para ello, el diseño cuenta con los siguientes bloques o cores IP:

- **2 VDMA (Video Direct Memory Access):** estos bloques están diseñados para proporcionar acceso directo a la memoria entre un stream de tipo AXI4 (Propietario de Xilinx) y un periférico de vídeo. En este caso, hay uno para la salida por HDMI y otro para la salida por VGA.
- **2 AXI Display Controller:** Encargado de mostrar la imagen de salida usando para ello sus tres componentes de color RGB (Rojo, verde y azul) así como de las señales de sincronización de vídeo horizontal y vertical. A su vez, se utiliza otro para dar las señales necesarias a un bloque adicional encargado de transformar las señales RGB a las necesarias para mostrar la imagen por la interfaz del HDMI.
- **1 Core propietario** con el que se generan las muestras para el audio.
- **3 cores de tipo AXI GPIO** usados para entradas/salidas digitales, en este caso 1 core para los switches, otro para los LEDs y otro para los botones de la Zybo.
- Por supuesto, el **core que controla el procesador de la placa**, y un par de cores necesarios para la interconexión de todos los AXI (**AXI Interconnect** y **AXI Protocol Converter**).

Sin embargo, en este diseño no se cumplen todas las necesidades que mi proyecto buscaba. Entre ellas, la más importante la captura del vídeo a través del HDMI.

Para conseguir un core que pudiera realizar esta función, se realizaron distintas búsquedas de información de diseños de referencia, documentación del puerto HDMI y toda clase de datos que pudieran serme de ayuda. Finalmente, se localizó un diseño que realiza algo similar a lo que se quiere conseguir en la web de Instructables, el cual se comentará en el apartado 3.1.3.

### 3.1.3 Image Filtering Demo + GoPRo

<sup>[INS, 15]</sup>Este diseño, estaba programado con el software (también de Xilinx): Xilinx Platform Studio EDK, el cual no es compatible con Vivado. Consecuentemente, de forma previa se llevó a cabo una migración del core útil para el diseño, un bloque dedicado exclusivamente a obtener datos de vídeo del puerto HDMI y ofrecer a su salida, un Stream de vídeo, o bien, los 3 canales de color por separado.

Tras varias pruebas, y algunos ajustes realizados al código VHDL del core, se consigue importar a la versión 2014.4 de Vivado el core, consiguiendo el siguiente bloque:

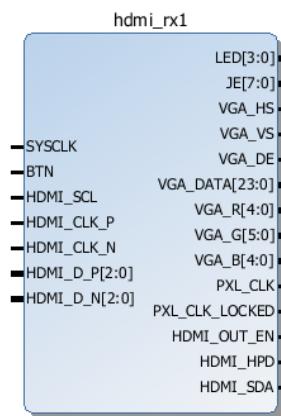


Ilustración 36. Core IP HDMI RX

Con este core, se puede directamente obtener un passthrough de vídeo entre el HDMI y el VGA, usando para ello las 3 salidas de color RGB conectadas a los pines específicos de la placa. Así, se puede visualizar directamente el vídeo procedente de la entrada HDMI por la salida VGA sin realizar ningún procesamiento adicional:

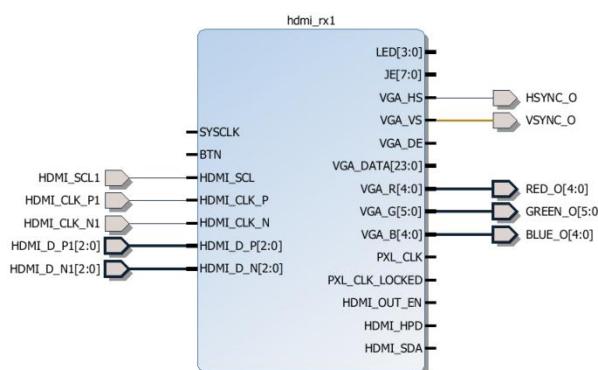


Ilustración 37. Conexionado para puente HDMI-VGA

Con este conexionado, se consigue directamente obtener la imagen desde el HDMI hacia la salida VGA del puerto de la Zybo configurando el fichero de restricciones (Señales) con Vivado.

## Procesado de vídeo con Zybo Zynq

```
##VGA Connector
##IO_L7P_T1_AD2P_35
set_property PACKAGE_PIN M19 [get_ports {RED_O[0]}]
set_property IOSTANDARD LVCMS33 [get_ports {RED_O[0]}]

##IO_L9N_T1_DQS_AD3N_35
set_property PACKAGE_PIN L20 [get_ports {RED_O[1]}]
set_property IOSTANDARD LVCMS33 [get_ports {RED_O[1]}]

##IO_L17P_T2_AD5P_35
set_property PACKAGE_PIN J20 [get_ports {RED_O[2]}]
set_property IOSTANDARD LVCMS33 [get_ports {RED_O[2]}]

##IO_L18N_T2_AD13N_35
set_property PACKAGE_PIN G20 [get_ports {RED_O[3]}]
set_property IOSTANDARD LVCMS33 [get_ports {RED_O[3]}]

##IO_L15P_T2_DQS_AD12P_35
set_property PACKAGE_PIN F19 [get_ports {RED_O[4]}]
set_property IOSTANDARD LVCMS33 [get_ports {RED_O[4]}]
```

Código 3. Definición puertos canal ROJO.

```
##IO_L14N_T2_SRCC_34
set_property PACKAGE_PIN P20 [get_ports {BLUE_O[0]}]
set_property IOSTANDARD LVCMS33 [get_ports {BLUE_O[0]}]

##IO_L7N_T1_AD2N_35
set_property PACKAGE_PIN M20 [get_ports {BLUE_O[1]}]
set_property IOSTANDARD LVCMS33 [get_ports {BLUE_O[1]}]

##IO_L10P_T1_AD11P_35
set_property PACKAGE_PIN K19 [get_ports {BLUE_O[2]}]
set_property IOSTANDARD LVCMS33 [get_ports {BLUE_O[2]}]

##IO_L14P_T2_AD4P_SRCC_35
set_property PACKAGE_PIN J18 [get_ports {BLUE_O[3]}]
set_property IOSTANDARD LVCMS33 [get_ports {BLUE_O[3]}]

##IO_L18P_T2_AD13P_35
set_property PACKAGE_PIN G19 [get_ports {BLUE_O[4]}]
set_property IOSTANDARD LVCMS33 [get_ports {BLUE_O[4]}]
```

Código 2. Definición puertos canal AZUL.

```
##IO_L14N_T2_AD4N_SRCC_35
set_property PACKAGE_PIN H18 [get_ports {GREEN_O[0]}]
set_property IOSTANDARD LVCMS33 [get_ports {GREEN_O[0]}]

##IO_L14P_T2_SRCC_34
set_property PACKAGE_PIN N20 [get_ports {GREEN_O[1]}]
set_property IOSTANDARD LVCMS33 [get_ports {GREEN_O[1]}]

##IO_L9P_T1_DQS_AD3P_35
set_property PACKAGE_PIN L19 [get_ports {GREEN_O[2]}]
set_property IOSTANDARD LVCMS33 [get_ports {GREEN_O[2]}]

##IO_L10N_T1_AD11N_35
set_property PACKAGE_PIN J19 [get_ports {GREEN_O[3]}]
set_property IOSTANDARD LVCMS33 [get_ports {GREEN_O[3]}]

##IO_L17N_T2_AD5N_35
set_property PACKAGE_PIN H20 [get_ports {GREEN_O[4]}]
set_property IOSTANDARD LVCMS33 [get_ports {GREEN_O[4]}]

##IO_L15N_T2_DQS_AD12N_35
set_property PACKAGE_PIN F20 [get_ports {GREEN_O[5]}]
set_property IOSTANDARD LVCMS33 [get_ports {GREEN_O[5]}]
```

Código 4. Definición puertos canal VERDE.

[ZAT, 15] Donde, como se puede apreciar, el canal verde utiliza 6 bits, mientras que los canales rojo y azul utilizan solamente 5. Esto se debe a que en la retina del ojo humano, los conos, encargados de la captura de los colores, son más sensibles a la luz verde/amarilla que a tonalidades rojas y azules.

Ya que en este proyecto se busca la captura del vídeo por HDMI y realizar un procesado del mismo, es necesario tener la señal completa con los 3 canales de color. Así por tanto se usa la señal VGA\_DATA de 24 bits con la que se obtiene la imagen capturada por HDMI completa.

Se utilizó como guía para obtener la imagen de entrada el conexionado del proyecto original:

```
##IO_L13N_T2_MRCC_34
set_property PACKAGE_PIN P19 [get_ports HSYNC_O]
set_property IOSTANDARD LVCMS33 [get_ports HSYNC_O]

##IO_O_34
set_property PACKAGE_PIN R19 [get_ports VSYNC_O]
set_property IOSTANDARD LVCMS33 [get_ports VSYNC_O]
```

Código 1. Definición puertos SINCRONISMO.

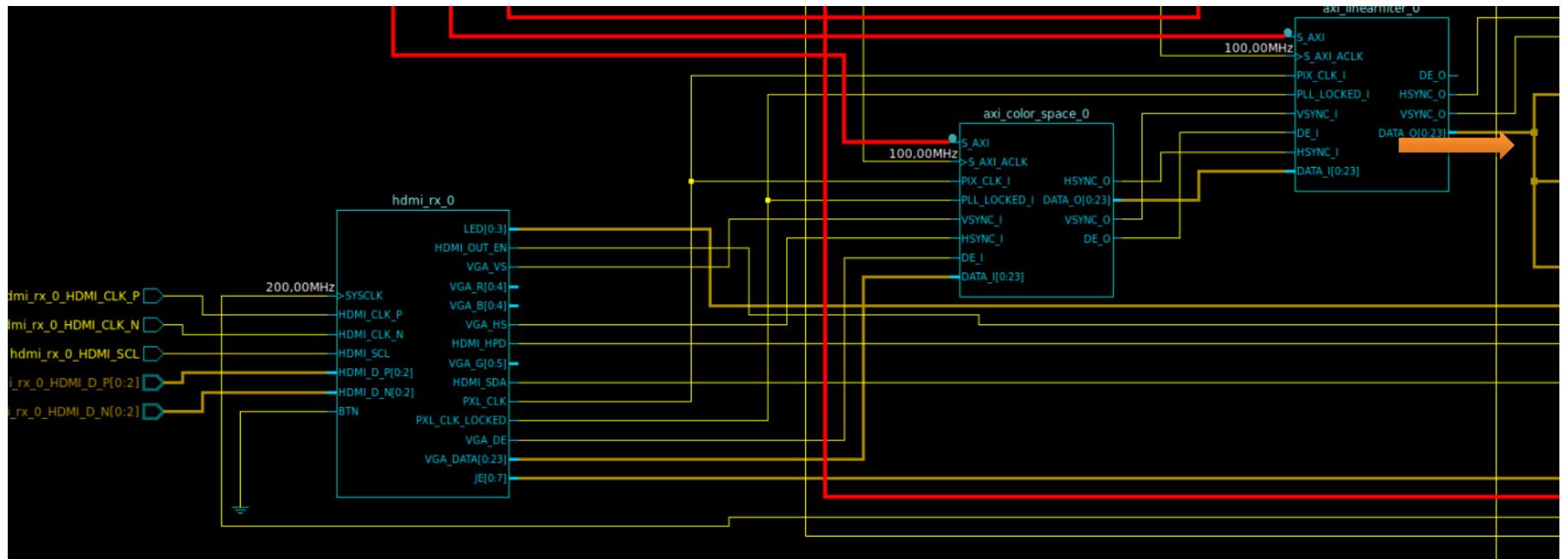


Ilustración 7. Diseño GoPro Instructables (1). Entradas.



Ilustración 8. Diseño GoPro Instructables. (2). Salidas.

Donde, como podemos ver, el stream de 24 bits se divide para obtener, en cada uno de los bus\_split, el color deseado para las salidas VGA.

### **3.2 Diseño propio. Procesado por software con 2 VDMAs.**

Una vez introducido todo lo necesario para mi diseño, veamos el diseño propio realizado para el procesado de vídeo, en un primer lugar, mediante software.

Para este diseño, se parte de una base en la que se utilizan 2 VDMAs donde, uno se utiliza para escribir en la memoria y otro para leer de la misma. A estos VDMAs los he nombrado respectivamente como VDMA\_HDMI y VDMA\_VGA pues, serán los encargados de:

- VDMA\_HDMI → Leer desde HDMI y escribir en memoria.
- VDMA\_VGA → Leer desde memoria y escribir en el VGA.

Por otra parte, tenemos el ya mencionado HDMI\_RX para obtener la imagen desde el puerto HDMI, y el AXI Display Controller para mostrar la imagen por el VGA. Todas estas conexiones, pueden verse en la ilustración 38:

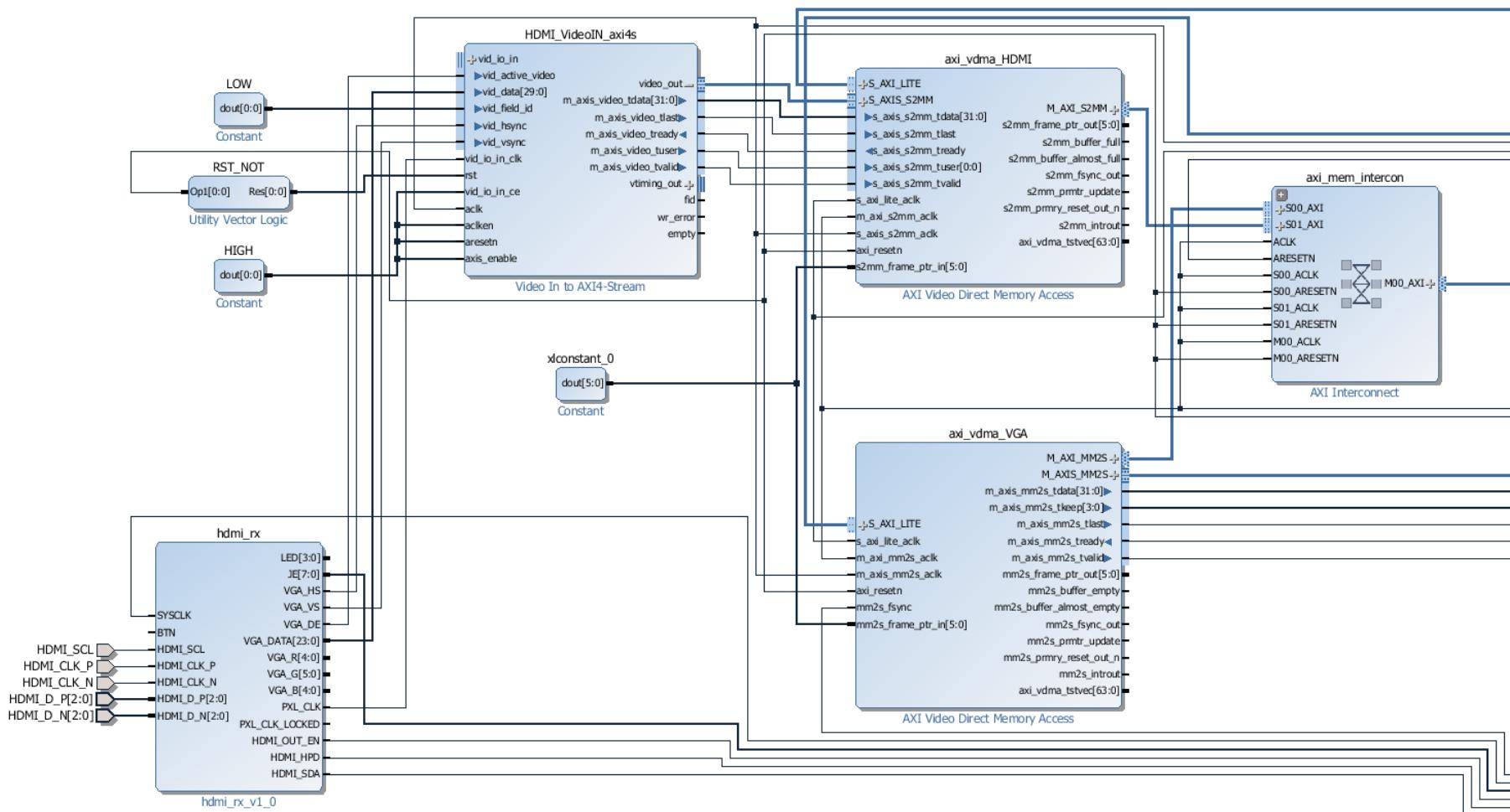


Ilustración 38. Detalle del diseño por software (1)

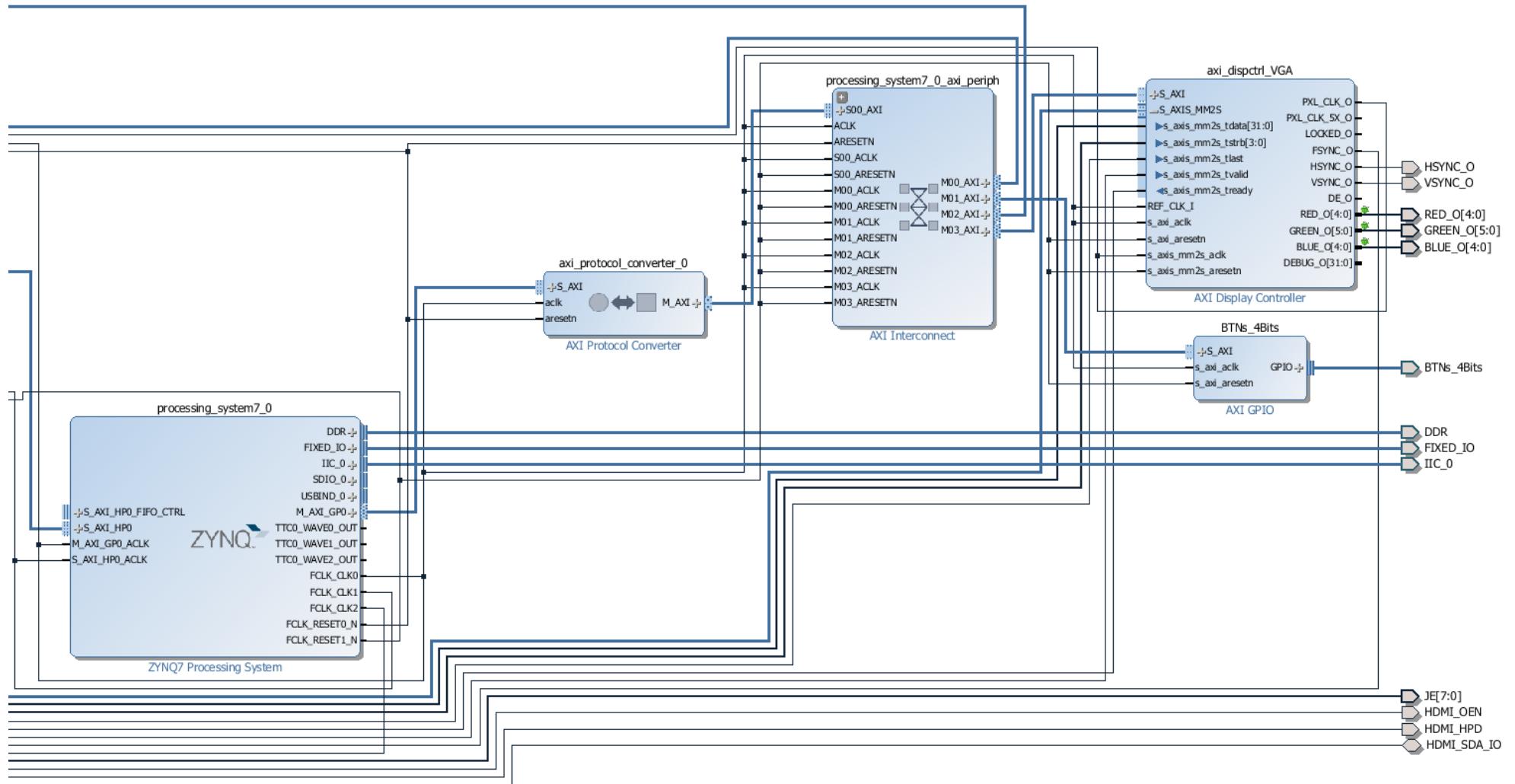
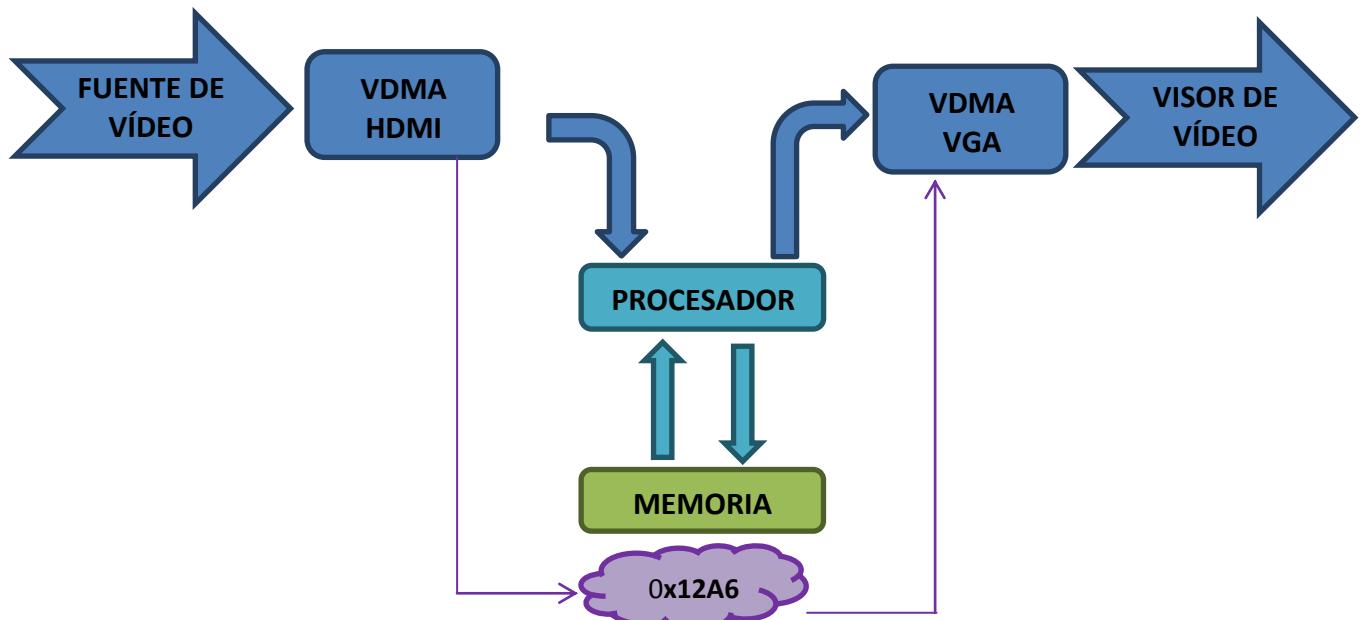


Ilustración 39. Detalle del diseño por software (2)

En el primer diseño, se programa de forma que el VDMA de escritura en memoria (VDMA HDMI) escriba en una dirección de memoria, en la cual el VDMA de lectura desde memoria (VDMA VGA) hará su lectura.

En el siguiente esquema, se muestra el flujo de los datos tanto de lectura como de escritura realizada por los VDMAs de una forma sencilla y visual para comprender el funcionamiento de este primer diseño:



Esquema 1. Diseño por software 1

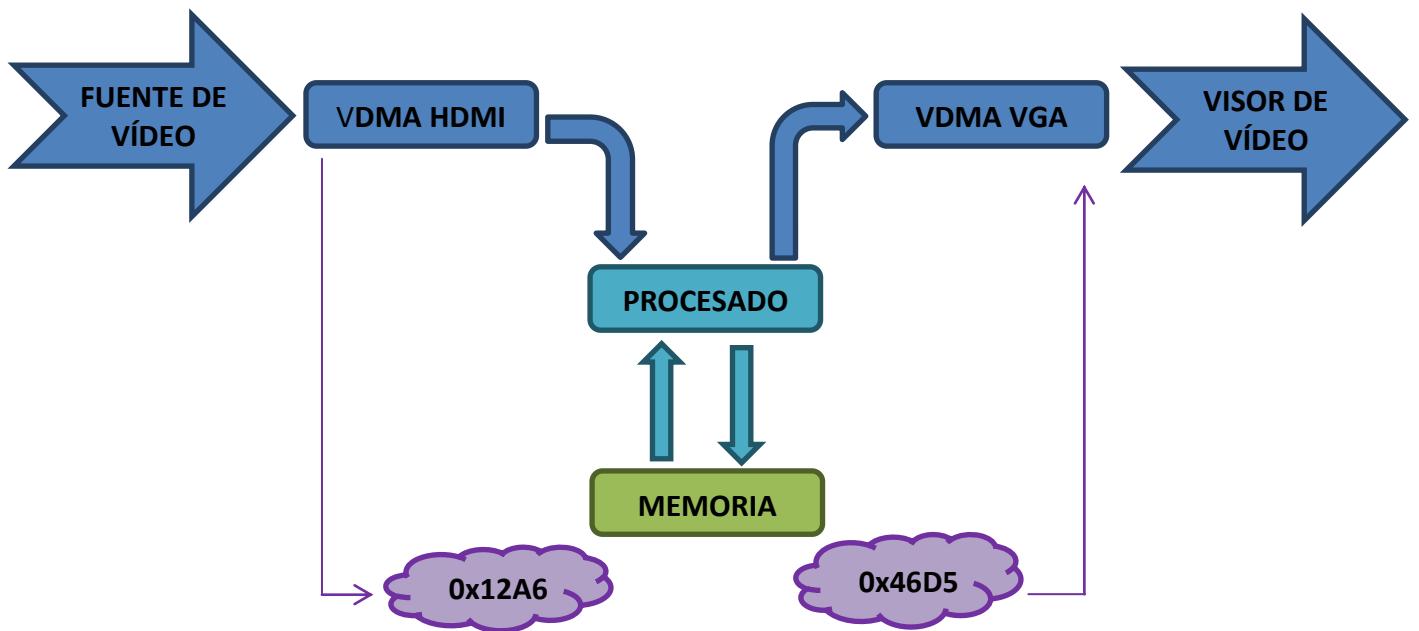
De esta manera, se consigue que la lectura de los datos, se haga de forma inmediata, sin ningún retardo apreciable. Para ello, los VDMAs deben configurarse usando la misma dirección de lectura/escritura (Según el uso que tenga el VDMA que se está configurando).

De esta forma, tenemos configurados ambos VDMAs, utilizando para los 2 el mismo *FrameStoreStartAddr* (Revisar punto 4.3), es decir, la misma dirección de lectura y escritura.

El VDMA HDMI escribe en memoria y lo realizará en una dirección, la cual el VDMA VGA leerá y mostrará por pantalla siendo para ambos la misma dirección.

En el segundo diseño de procesado software, la diferencia respecto al primero, es que el VDMA HDMI escribe en una dirección de memoria, mientras que el VDMA VGA lee de otra dirección de memoria diferente.

En el esquema presentado a continuación, se muestra de nuevo el flujo de datos de lectura y escritura entre VDMAs con la diferencia respecto al primer diseño de que cada VDMA utiliza una **dirección de memoria diferentes**.



Esquema 2. Diseño por software 2

Es por esto, que en este diseño, es necesario una actuación dentro del código diseñado para manejar el hardware, donde se haga constantemente un copiado de los datos en la dirección “0x12A6” a la dirección “0x46D5”.

Esto, implica una latencia entre el vídeo de la entrada y la visualización a la salida. Para estas pruebas se han utilizado 2 formas diferentes de copiado de datos:

1. *Copiado mediante un bucle for:*

Utilizando para ello el siguiente código:

```

while(1){
    for (i=0; i < DISPLAY_NUM_FRAMES; i++){
        for (j=0; j < DISPLAYDEMO_MAX_FRAME ; j++){
            *(vgaPtr[i]+j) = *(hdmiPtr[i]+j);
        }
    }
}
  
```

Código 5. Copiado mediante bucle for

Donde vgaPtr y hdmiPtr son, respectivamente, las direcciones de lectura y escritura de los VDMAs, DISPLAY\_NUM\_FRAMES es el número de frames configurados (3) y DISPLAY\_MAX\_FRAME indica la resolución usada: 1280x720 píxeles .

El uso de un while(1) implica tener constantemente la Zybo conectada al. SDK para que éste ejecute el proceso de copiado de las direcciones de memoria, provocando una constante ejecución en el procesador.

Además, el hecho de tener que recorrer: 1280\*720\*3 píxeles e ir copiando uno a uno, llegaba a añadir una latencia en la salida del vídeo de 3 segundos, lo que hace inviable su uso.

## 2. Copiado mediante bloques de memoria con “memCpy”:

Utilizando este método, se consigue una notable mejora respecto a la opción 1, pero sigue visualizándose un retardo de entre unos 300 – 400 ms. Para este método, el código sería el siguiente:

```
while(1){  
    for(i = 0; i < DISPLAY_NUM_FRAMES; ++ i){  
  
        memcpy(&(vgaPtr[i][0]), &(hdmiPtr[i][0]), DISPLAYDEMO_MAX_FRAME * sizeof(u32));  
    }  
}
```

Código 6. Copiado mediante "memCpy"

## 3.3 Diseño propio. Procesado por hardware con 3 VDMAs.

En este diseño, se utiliza la arquitectura óptima para el funcionamiento del procesado de vídeo, utilizando para ello 3 VDMAs que se encargarán de:

- VDMA\_HDMI → Encargado de recoger la señal de vídeo de entrada por el puerto HDMI, y escribirlo en memoria.
- VDMA\_VGA → Encargado de leer la señal de vídeo de memoria, y escribirlo por el puerto VGA para su visualización.
- VDMA\_HLS → Un VDMA que tendrá a su vez un puerto de escritura en memoria y otro de lectura de la misma. Este VDMA será el encargado de recoger el vídeo escrito en memoria por el VDMA\_HDMI para enviarlo al core generado por HLS para el procesado del vídeo y después, de recoger el vídeo procesado por el core y escribirlo de nuevo en la memoria para que el VDMA\_VGA pueda leerlo y mostrarlo por pantalla.

De modo, que el diseño en Vivado queda así:

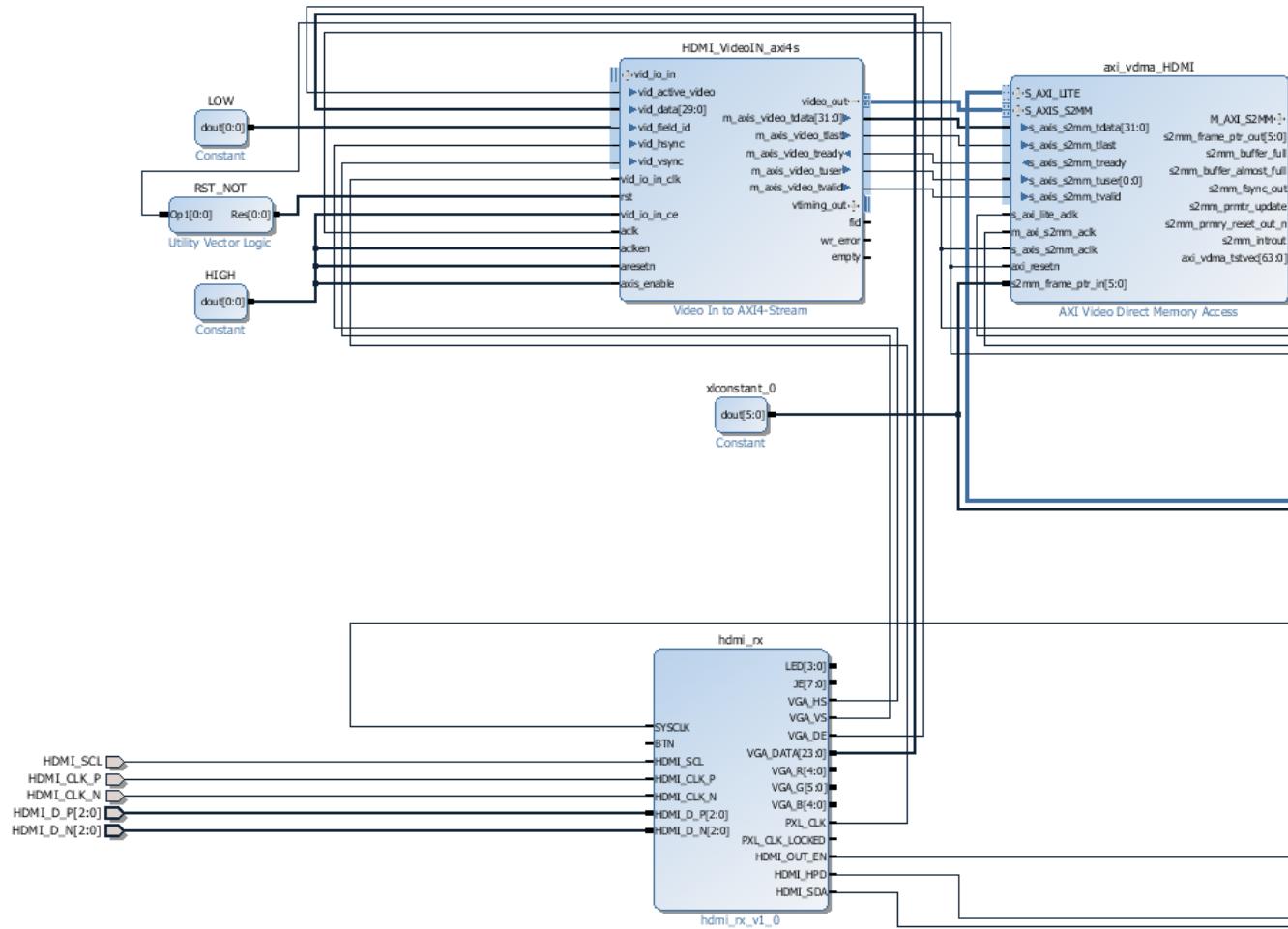


Ilustración 40. Detalle del diseño por Hardware (1)

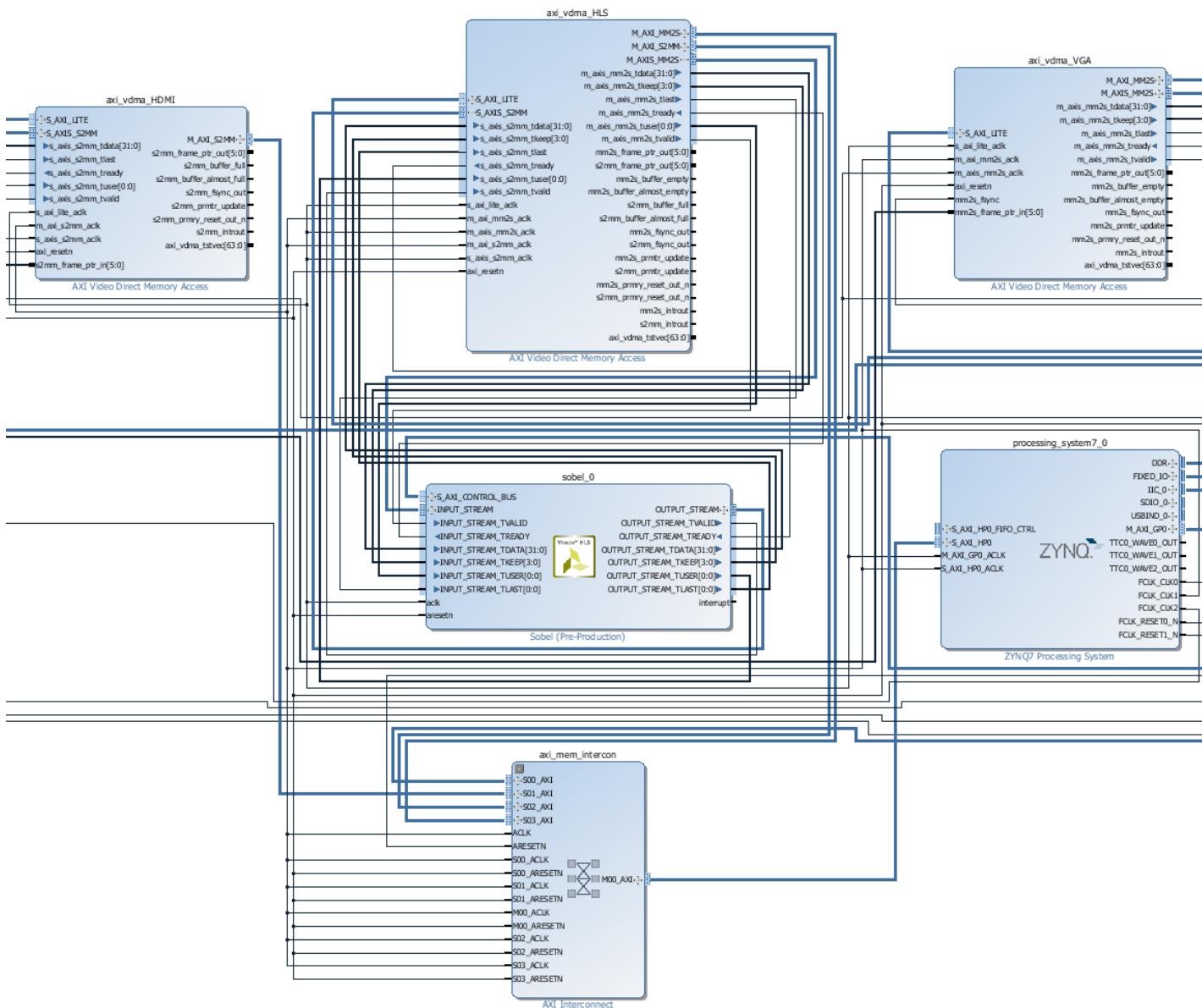


Ilustración 41. Detalle del diseño por Hardware (2)

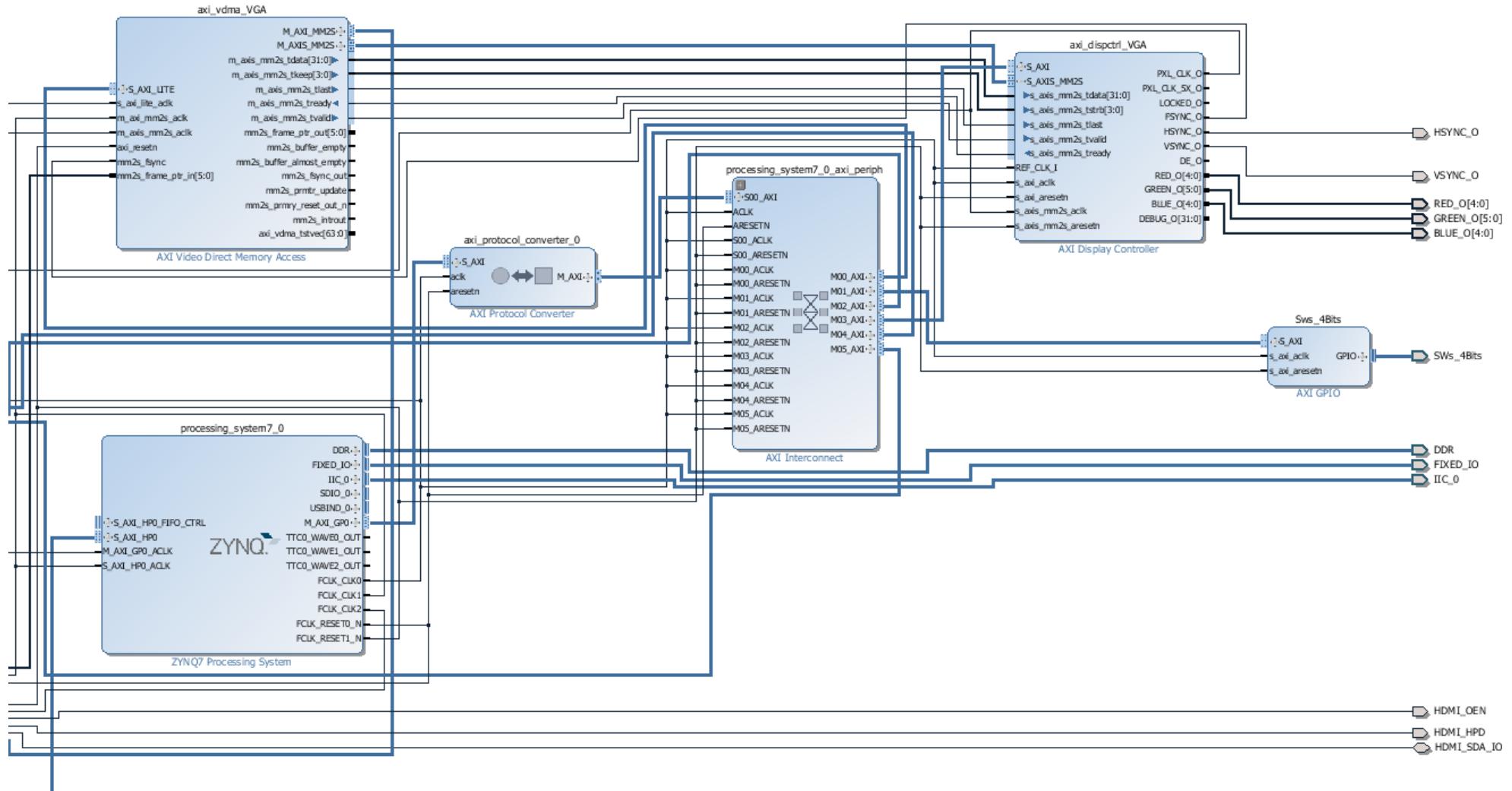
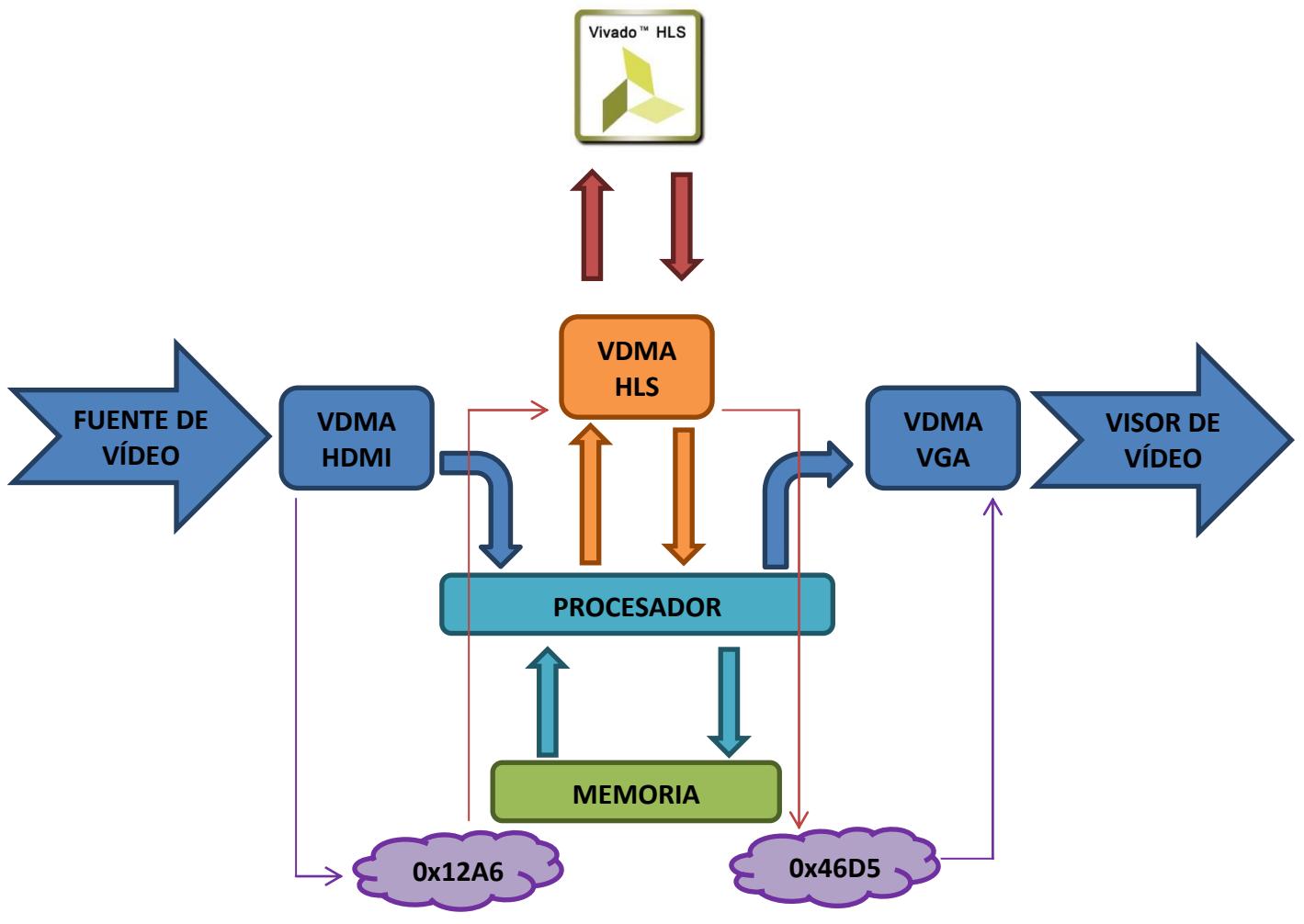


Ilustración 42. Detalle del diseño por Hardware (3)

Donde, si nos fijamos en la ilustración 14, podemos ver la interconexión entre el VDMA\_HLS que incorpora tanto lectura como escritura, y el core HLS del procesado, denominado “Sobel\_0” y que las conexiones tanto de lectura como de escritura, se realizan entre ambos.

Este es el esquema funcional de este diseño, en el que cada VDMA utiliza una dirección de memoria diferente y como core intermedio está el core HLS que realiza el procesamiento del video. El VDMA HDMI escribe en la dirección de memoria A, la cual se lee desde el VDMA HLS y a continuación el VDMA HLS escribe en una dirección de memoria B la cual se lee desde el VDMA VGA completando todo el proceso:



De modo que, en este caso, el vídeo capturado por el VDMA\_HDMI, se lee por el VDMA\_HLS para que se utilice como entrada en el core de procesamiento de vídeo, realizado con Vivado HLS. Éste a su vez, realiza el procesamiento definido en su hardware, y lo devuelve ya procesado.

De nuevo, el VDMA\_HLS escribe en memoria el vídeo con el procesado realizado, y es entonces cuando el VDMA\_VGA lee la memoria y muestra por el dispositivo de salida el vídeo ya procesado, obteniendo por tanto un tratamiento de

vídeo en tiempo real, sin retardo alguno, y pudiendo así añadir nuevas funcionalidades al core HLS.

Pero... ¿Cómo funciona el Core HLS?

### 3.3.1 Programación y diseño del Core HLS de procesamiento de vídeo.

Vivado HLS es una herramienta muy potente y sencilla de utilizar, cuyo cometido es convertir código de alto nivel, como C o C++, a un código de bajo nivel como VHDL o Verilog con el añadido de que es un código sintetizable, al que le podemos añadir ciertas directrices consiguiendo un diseño más eficiente en función de los requisitos que tenga nuestro diseño:

- Espacio de memoria
- Potencia de la placa
- Requisitos de tiempos máximos de ejecución
- Etc...

En una primera instancia, el diseño se iba a programar utilizando para ello las herramientas que ofrece OpenCV<sup>[XIL, 15, A]</sup> ya que, es una herramienta compatible con Vivado HLS, pero presenta un inconveniente: OpenCV necesita de un sistema operativo para funcionar.

El objetivo de este proyecto era conseguir hacer un sistema autónomo de tipo stand-alone, es decir, que no precisara de un sistema operativo.

Xilinx, dispone de unas herramientas de procesamiento del vídeo, similares a OpenCV, denominadas “HLS Vídeo Library”<sup>[WIK, 15, B]</sup> las cuales, se incluyen dentro de las librerías ofrecidas por Vivado HLS y son funciones sintetizables para acelerar las aplicaciones de procesado de imagen.

La librería de HLS Vídeo son funciones de procesado de vídeo, muy similares a OpenCV, entre las cuales, podemos encontrar:

- Detección de esquinas de Harris.
- Filtros de erosión y dilatación.
- Filtrados 2D.
- Filtrados gaussianos.
- Inversión del color y la imagen.
- Detección de bordes de Sobel.
- Redimensión de la imagen.
- Etc...

Las cuales, están documentadas, tanto su funcionalidad como su integración en el código, en la Wiki de Xilinx, en la dirección web de la referencia [WIK, 15, B].

El core realizado para este diseño es muy sencillo y simple, pero a su vez potente y eficiente. En resumen, el core:

1. Recibe un stream de vídeo de tipo AXI\_STREAM, el cual se transforma al tipo “MAT” que es con el que se puede trabajar usando las funciones propias de sintetización de HLS Vídeo, almacenándose en otra variable intermedia de

tipo MAT con la resolución máxima permitida, que en este caso es de 1280x720 píxeles.

2. Se le aplica un procesado, en este caso un filtrado de Sobel para detección de bordes, utilizando para ello la función propia de HLS Vídeo, ya que es una función sintetizable, consiguiendo una mejor optimización en el proceso y que además, como ya se ha mencionado, no precisa de usar un sistema operativo como ocurre con OpenCV.

Sin embargo, el filtrado de Sobel, se realiza aplicando antes varios procesos, pues ya que al Core HLS llega una imagen en color, y Sobel se utiliza sobre un solo canal monocromo. Se aplica primero una transformación para convertir de RGB a escala de grises, utilizando para ello la función “CvtColor”.

A continuación, se realiza un duplicado de la imagen en blanco y negro, pues Vivado HLS no permite realizar más de una tarea sobre una misma imagen. Por esto, se utiliza la función “Duplicate”.

Con esto se le está indicando al hardware, que se está realizando una división del flujo de datos, ya que de no hacerlo de esta forma, se estaría escribiendo y leyendo al mismo tiempo de una zona de memoria para dos operaciones diferentes.

Después, se aplica un filtrado de Sobel vertical a una de las imágenes creadas con “Duplicate” y un filtrado horizontal a la otra, realizando la suma de ambas para conseguir la imagen final.

Para terminar, a esta imagen final se le aplica de nuevo un cambio en la escala de color, de escala de grises, a RGB aunque es imposible obtener color desde una imagen en escala de grises, debe pasarse a RGB ya que de lo contrario, el controlador recibirá una imagen que no es la que espera, devolviendo un error. La otra opción, sería utilizar la imagen monocromo triplicando los mismos datos en cada uno de los tres canales RGB.

3. Se realiza la transformación inversa del tipo MAT (última imagen con filtrado realizado) al tipo AXI\_STREAM para la salida del vídeo.

El código, sería el siguiente:

```
#include <hls_video.h>
#include "sobel.h"
//Puedo crear un objeto de imagen con:
hls::Mat<720, 1280, HLS_8UC3> imagenHLS();

void sobel(AXI_STREAM& INPUT_STREAM, AXI_STREAM& OUTPUT_STREAM, int rows, int cols) {

#pragma HLS INTERFACE axis port=INPUT_STREAM
#pragma HLS INTERFACE axis port=OUTPUT_STREAM
#pragma HLS RESOURCE core=AXI_SLAVE variable=rows metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=cols metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=return metadata="-bus_bundle CONTROL_BUS"
#pragma HLS INTERFACE ap_stable port=rows
#pragma HLS INTERFACE ap_stable port=cols
```

Código 7. Core HLS Sobel (1). Directrices para interfaces

```
RGB_IMAGE img_0(rows, cols);
GRAY_IMAGE img_1(rows, cols);
GRAY_IMAGE img_2(rows, cols);
GRAY_IMAGE img_3(rows, cols);
GRAY_IMAGE img_4(rows, cols);
GRAY_IMAGE img_5(rows, cols);
GRAY_IMAGE img_6(rows, cols);
RGB_IMAGE img_7(rows, cols);

#pragma HLS dataflow
```

Código 8. Core HLS Sobel (2). Declaración de variables intermedias necesarias

```
//Se convierte el stream de entrada, de un AXIVideo al formato MAT, y se almacena en img_0
hls::AXIVideo2Mat(INPUT_STREAM, img_0);

//A continuación, se hace una conversión de color de RGB a escala de grises
hls::CvtColor<HLS_RGB2GRAY>(img_0, img_1);

//Para poder hacer un sobel vertical y luego uno horizontal a la misma imagen, se hace un duplicado de img_1 en img_2
// e img_3
hls::Duplicate(img_1,img_2,img_3);

//Se aplica sobel vertical a img_2 y se guarda en img_4
hls::Sobel<0,1,3>(img_2, img_4);

//Se aplica Sobel horizontal a img_3 y se guarda en img_5
hls::Sobel<1,0,3>(img_3, img_5);

//Finalmente, se suman ambos procesados en img_6
hls::AddWeighted(img_4,1,img_5,1,img_6);

//Y se hace una conversión de nuevo de escala de gris, a RGB puesto que es lo que espera el controlador del core
hls::CvtColor<HLS_GRAY2RGB>(img_6, img_7);

//Y finalmente, transformamos la img_7 de MAT a AXIVideo, y se pega en OUTPUT_STREAM
hls::Mat2AXIVideo(img_7, OUTPUT_STREAM);
```

Código 9. Core HLS Sobel (3). Distintas funciones de filtrado de imagen

En los ficheros incluidos en el código, tenemos el “`hls_video.h`” que nos proporciona todo lo necesario para el manejo de los streams de vídeo, y el “`sobel.h`” el cual, incluye la definición de los tipos Stream, Mat y Scalar que se utilizan en el código del core, así como la definición de la función TOP que es como el “main” en HLS, es decir, el código principal que ejecutará el core.

```
#ifndef _TOP_H_
#define _TOP_H_

#include "hls_video.h"

// maximum image size
#define MAX_WIDTH 1280
#define MAX_HEIGHT 720

// Se definen las estructuras.
typedef hls::stream<ap_axiu<32,1,1,1>> AXI_STREAM;
typedef hls::Scalar<3, unsigned char> RGB_PIXEL;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC3> RGB_IMAGE; //Imagen en color
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC1> GRAY_IMAGE; //Imagen en escala de gris

// top level function for HW synthesis
void sobel(AXI_STREAM& INPUT_STREAM, AXI_STREAM& OUTPUT_STREAM, int rows, int cols);

#endif
```

Código 10. "Sobel.h" incluido en el TOP del core

Una vez diseñado el core se realiza la síntesis del código C de la solución activa. Vivado HLS permite tener dentro de la misma programación del core distintas soluciones (propuestas de código para sintetizar) de manera que podamos realizar varias dentro del mismo proyecto evitando así tener duplicados.

Con esto, podemos tener el mismo código, es decir, realizar la misma tarea pero añadir o modificar las directrices creadas, para mejorar u optimizar los requisitos del diseño.

En este caso, y por simplicidad, sólo se tiene una solución implementada en el proyecto.

Para realizar la síntesis, teniendo el código principal que contiene la función TOP del proyecto abierto en la pantalla principal, seleccionamos el menú “*Solution → Run C Synthesis → Active Solution*”

A continuación, se muestra la interfaz gráfica de Vivado HLS y una pequeña descripción de las distintas ventanas del entorno de desarrollo:

## Procesado de vídeo con Zybo Zynq

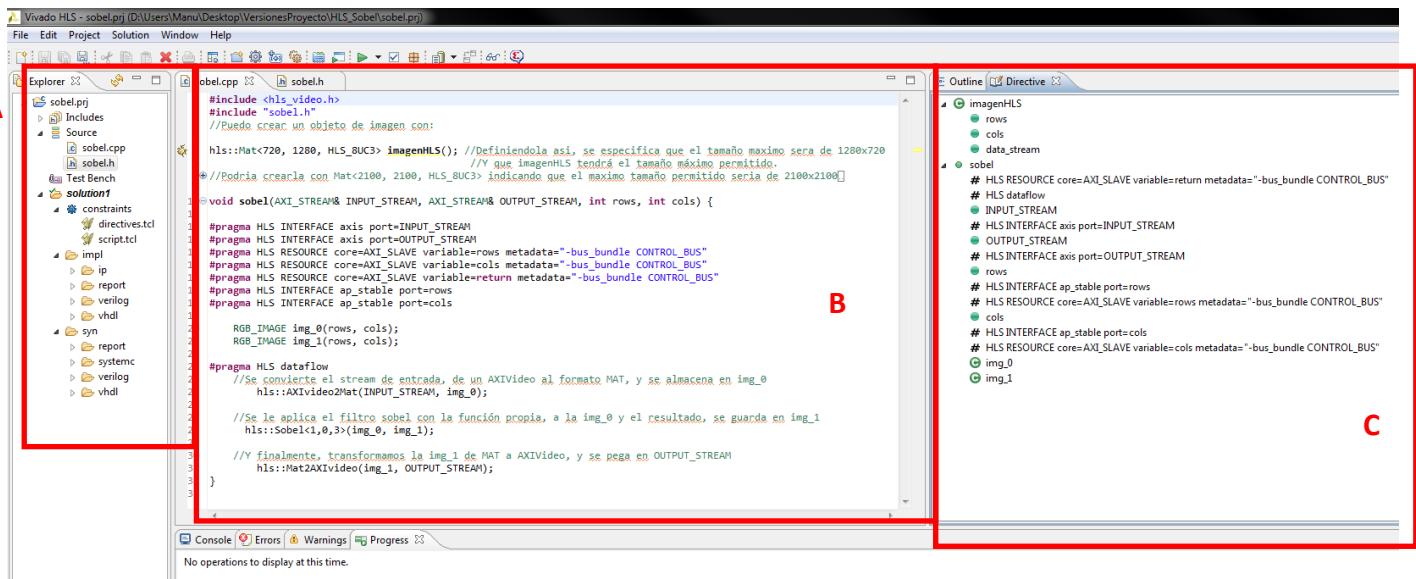


Ilustración 43. Ventana principal Vivado HLS

**A. Explorador del proyecto:** Aquí podemos navegar entre los distintos directorios que ofrece el proyecto. Los más importantes son el “Source” que incluye el código fuente que se sintetizará en el core. El fichero que contenga la función TOP, debe tener el mismo nombre que se especifica para el proyecto al crearlo nuevo.

**B. Pantalla principal de Vivado HLS:** Aquí se visualizan los distintos ficheros de código que se pueden modificar y añadir al proyecto. Es importante que el código top de la función, esté abierto cuando se vaya a sintetizar la solución.

**C. Directivas del diseño:** En esta ventana se pueden ver las directivas creadas para el diseño, con el que se crean tanto los puertos de entrada/salida que se necesiten, como opciones creadas para requisitos de tiempo o espacio en la placa destino.

Una vez que se tiene todo el código realizado y se quiere comprobar la solución, hay que realizar la síntesis C de la solución activa.

Para realizar la síntesis mencionada, se elige el menú “Solution” de Vivado HLS y marcando dentro del submenú “Run C Synthesis → Active Solution” teniendo en la pantalla principal de Vivado HLS el código TOP del diseño.

Se muestra el proceso en la imagen siguiente:

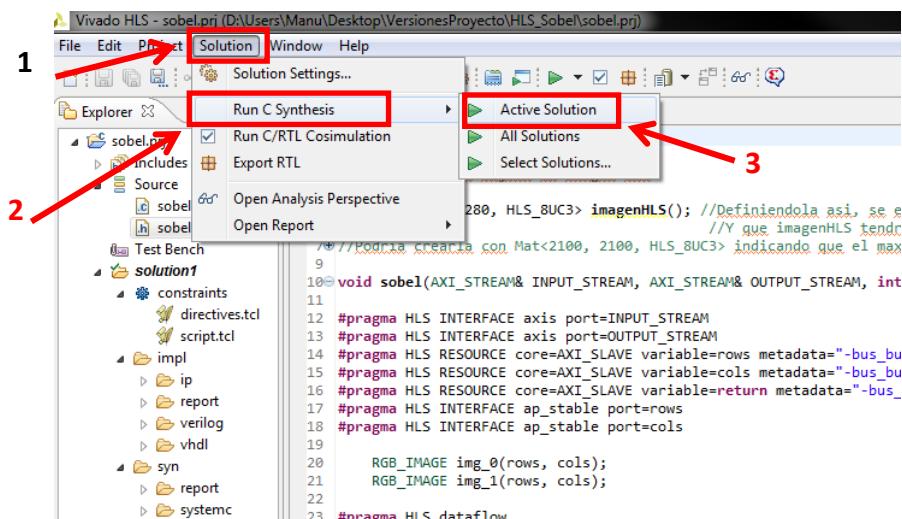


Ilustración 44. Iniciar síntesis de la solución

Una vez que se realiza la síntesis de la solución, se generan nuevos directorios en el explorador del proyecto. En particular, se genera un directorio denominado “Syn”:

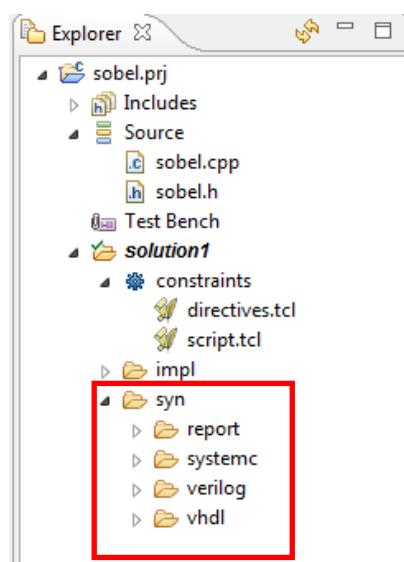


Ilustración 45. Directorio SYN creado tras la síntesis.

Dentro de este directorio, Vivado HLS ofrece un informe detallado de la síntesis realizada, indicando estimaciones del rendimiento del core, así como códigos en

SystemC, VHDL y Verilog que se utilizarán más adelante por el propio Vivado HLS para realizar la implementación del Core.

Dentro de los informes, tenemos:

- ***Estimaciones de tiempo:***

Clock	Target	Estimated	Uncertainty
default	8.00	6.86	1.00

Ilustración 46. Resumen de tiempos de ejecución

- ***Latencia (Resumen y detalle):***

Instance	Module	min	max	min	max	Type
call_ret_sobel_Block_proc_fu_217	sobel_Block_proc	0	0	0	0	none
grp_sobel_AXIvideo2Mat_fu_171	sobel_AXIvideo2Mat	3	924483	3	924483	none
grp_sobel_Sobel_fu_159	sobel_Sobel	154	935986	154	935986	none
grp_sobel_Mat2AXIvideo_fu_194	sobel_Mat2AXIvideo	1	923761	1	923761	none

Ilustración 47. Resumen y detalle de la latencia en la ejecución

- ***Estimación de la utilización del dispositivo:*** Comprobando la imagen siguiente, puede observarse que se utiliza sólo un 7% de los BlockRAMs disponibles en la placa y un 19% del total de LUTs además de que no se utilizan ninguno de los 80 DSPs que incluye.

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	3
FIFO	0	-	60	264
Instance	9	-	1656	3144
Memory	-	-	-	-
Multiplexer	-	-	-	12
Register	-	-	11	-
Total	9	0	1727	3423
Available	120	80	35200	17600
Utilization (%)	7	0	4	19

Ilustración 48. Resumen de la utilización

En la siguiente tabla, puede comprobarse qué core es el que más recursos utiliza y por tanto dónde podría realizarse una mejor optimización si fuera necesario.

□ **Detail**

□ **Instance**

Instance	Module	BRAM_18K	DSP48E	FF	LUT
sobel_AXIvideo2Mat_U0	sobel_AXIvideo2Mat	0	0	240	291
sobel_Block_proc_U0	sobel_Block_proc	0	0	74	73
sobel_Mat2AXIvideo_U0	sobel_Mat2AXIvideo	0	0	60	109
sobel_Sobel_U0	sobel_Sobel	9	0	1282	2671
Total		4	9	0	1656
					3144

Ilustración 49. Detalle de utilización de la instancia

□ **FIFO**

Name	BRAM_18K	FF	LUT	Depth	Bits	Size:D*B
img_0_cols_V_channel1_U	0	5	24	2	12	24
img_0_cols_V_channel_U	0	5	24	2	12	24
img_0_data_stream_0_V_U	0	5	20	1	8	8
img_0_data_stream_1_V_U	0	5	20	1	8	8
img_0_data_stream_2_V_U	0	5	20	1	8	8
img_0_rows_V_channel1_U	0	5	24	2	12	24
img_0_rows_V_channel_U	0	5	24	2	12	24
img_1_cols_V_U	0	5	24	2	12	24
img_1_data_stream_0_V_U	0	5	20	1	8	8
img_1_data_stream_1_V_U	0	5	20	1	8	8
img_1_data_stream_2_V_U	0	5	20	1	8	8
img_1_rows_V_U	0	5	24	2	12	24
Total	0	60	264	18	120	192

Ilustración 50. Detalle de la utilización FIFO

□ **Expression**

Variable Name	Operation	DSP48E	FF	LUT	Bitwidth P0	Bitwidth P1
sobel_AXIvideo2Mat_U0_ap_start	and	0	0	1	1	1
sobel_Mat2AXIvideo_U0_ap_start	and	0	0	1	1	1
sobel_Sobel_U0_ap_start	and	0	0	1	1	1
Total		3	0	0	3	3

Ilustración 51. Detalle de la utilización de la expresión

□ Multiplexer

Name	LUT	Input Size	Bits	Total Bits
ap_chn_write_sobel_Block_proc_U0_img_0_cols_V_channel	1	2	1	2
ap_chn_write_sobel_Block_proc_U0_img_0_cols_V_channel1	1	2	1	2
ap_chn_write_sobel_Block_proc_U0_img_0_rows_V_channel	1	2	1	2
ap_chn_write_sobel_Block_proc_U0_img_0_rows_V_channel1	1	2	1	2
ap_chn_write_sobel_Block_proc_U0_img_1_cols_V	1	2	1	2
ap_chn_write_sobel_Block_proc_U0_img_1_rows_V	1	2	1	2
ap_sig_ready_img_0_cols_V_channel1_full_n	1	2	1	2
ap_sig_ready_img_0_cols_V_channel_full_n	1	2	1	2
ap_sig_ready_img_0_rows_V_channel1_full_n	1	2	1	2
ap_sig_ready_img_0_rows_V_channel_full_n	1	2	1	2
ap_sig_ready_img_1_cols_V_full_n	1	2	1	2
ap_sig_ready_img_1_rows_V_full_n	1	2	1	2
Total	12	24	12	24

Ilustración 52. Detalle de la utilización del multiplexor

□ Register

Name	FF	LUT	Bits	Const Bits
ap_CS	1	0	1	0
ap_reg_procdone_sobel_AXIvideo2Mat_U0	1	0	1	0
ap_reg_procdone_sobel_Block_proc_U0	1	0	1	0
ap_reg_procdone_sobel_Mat2AXIvideo_U0	1	0	1	0
ap_reg_procdone_sobel_Sobel_U0	1	0	1	0
ap_reg_ready_img_0_cols_V_channel1_full_n	1	0	1	0
ap_reg_ready_img_0_cols_V_channel_full_n	1	0	1	0
ap_reg_ready_img_0_rows_V_channel1_full_n	1	0	1	0
ap_reg_ready_img_0_rows_V_channel_full_n	1	0	1	0
ap_reg_ready_img_1_cols_V_full_n	1	0	1	0
ap_reg_ready_img_1_rows_V_full_n	1	0	1	0
Total	11	0	11	0

Ilustración 53. Detalle de utilización de los registros

Y finalmente, ofrece también un informe de las interfaces que se crearán en el core que se va a generar:

Interface					
Summary					
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
INPUT_STREAM_TDATA	in	32	axis	INPUT_STREAM_V_data_V	pointer
INPUT_STREAM_TKEEP	in	4	axis	INPUT_STREAM_V_keep_V	pointer
INPUT_STREAM_TSTRB	in	4	axis	INPUT_STREAM_V_strb_V	pointer
INPUT_STREAM_TUSER	in	1	axis	INPUT_STREAM_V_user_V	pointer
INPUT_STREAM_TLAST	in	1	axis	INPUT_STREAM_V_last_V	pointer
INPUT_STREAM_TID	in	1	axis	INPUT_STREAM_V_id_V	pointer
INPUT_STREAM_TDEST	in	1	axis	INPUT_STREAM_V_dest_V	pointer
INPUT_STREAM_TVALID	in	1	axis	INPUT_STREAM_V_dest_V	pointer
INPUT_STREAM_TREADY	out	1	axis	INPUT_STREAM_V_dest_V	pointer
OUTPUT_STREAM_TDATA	out	32	axis	OUTPUT_STREAM_V_data_V	pointer
OUTPUT_STREAM_TKEEP	out	4	axis	OUTPUT_STREAM_V_keep_V	pointer
OUTPUT_STREAM_TSTRB	out	4	axis	OUTPUT_STREAM_V_strb_V	pointer
OUTPUT_STREAM_TUSER	out	1	axis	OUTPUT_STREAM_V_user_V	pointer
OUTPUT_STREAM_TLAST	out	1	axis	OUTPUT_STREAM_V_last_V	pointer
OUTPUT_STREAM_TID	out	1	axis	OUTPUT_STREAM_V_id_V	pointer
OUTPUT_STREAM_TDEST	out	1	axis	OUTPUT_STREAM_V_dest_V	pointer
OUTPUT_STREAM_TVALID	out	1	axis	OUTPUT_STREAM_V_dest_V	pointer
OUTPUT_STREAM_TREADY	in	1	axis	OUTPUT_STREAM_V_dest_V	pointer
rows	in	32	ap_stable	rows	scalar
cols	in	32	ap_stable	cols	scalar
ap_clk	in	1	ap_ctrl_hs	sobel	return value
ap_RST_n	in	1	ap_ctrl_hs	sobel	return value
ap_start	in	1	ap_ctrl_hs	sobel	return value
ap_done	out	1	ap_ctrl_hs	sobel	return value
ap_idle	out	1	ap_ctrl_hs	sobel	return value
ap_ready	out	1	ap_ctrl_hs	sobel	return value

Ilustración 54. Informe de las interfaces a generar

La elección de generar las señales como entradas o salidas se realiza de forma automática en función del código programado en la función TOP. Vivado HLS lo interpreta para realizar la implementación del core de la manera más eficiente con el objetivo de conseguir un mejor rendimiento.

Una vez realizada la síntesis de la solución activa, queda generar la implementación del core y hacer la exportación RTL del mismo. Con esto, Vivado HLS genera un fichero ZIP que incluye el core que se puede importar en el proyecto de Vivado, dentro del diseño de bloques, para finalmente realizar las conexiones con el resto del diseño e integrarlo para que realice su función.

Para implementar el core hay que seleccionar el menú “Solution” de Vivado HLS y a continuación pulsar la opción Export RTL. Se muestra el procedimiento en la siguiente imagen:

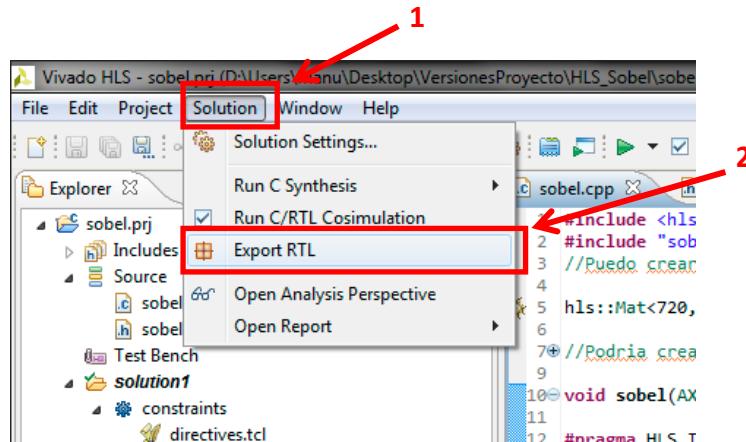


Ilustración 55. Exportar el RTL.

Al igual que al realizar la síntesis, tras realizar la implementación y la exportación del RTL, se generan nuevos directorios en el explorador. En concreto, se genera el directorio “impl” que incluye, entre otros, el directorio “ip” en el cual se genera el core que se importa en Vivado:

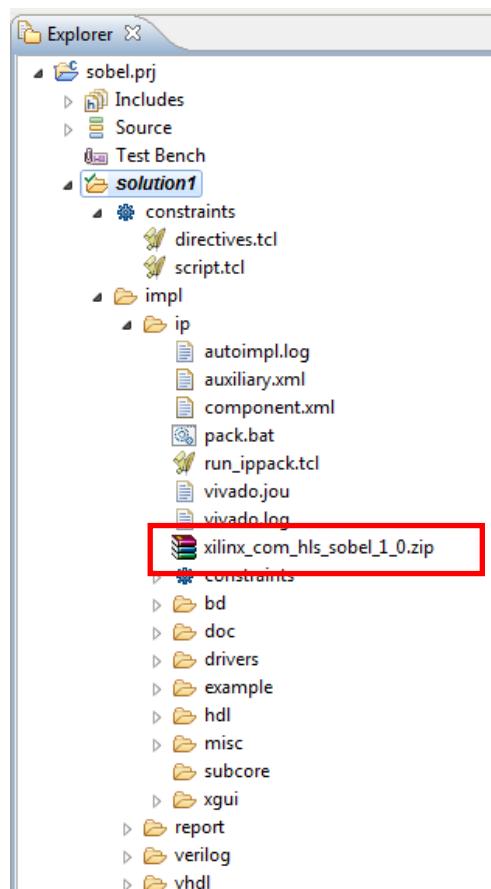


Ilustración 56. Directorio "impl" y Core a importar en Vivado

### 3.3.2 Configurando los VDMA

Una de las tareas más complejas que se han llevado a cabo en este proyecto, ha sido conseguir que los cores VDMA, funcionaran correctamente.

Partiendo del diseño base que ofrece el fabricante (Punto 4.1.2), se observa que la configuración se realiza de forma individualizada para cada uno de los VDMA.

Además, los VDMA deben tener una configuración en el hardware, que Vivado nos permite modificar directamente desde el diseño de bloques haciendo un doble clic sobre él.

La mejor manera de configurar un hardware propietario de Xilinx, es utilizar los recursos que la propia Xilinx ofrece ya que indican la forma más adecuada de configuración.

Para esto, una vez realizada la implementación del diseño en Vivado, desde el propio SDK, se crea un nuevo BSP de la siguiente forma:

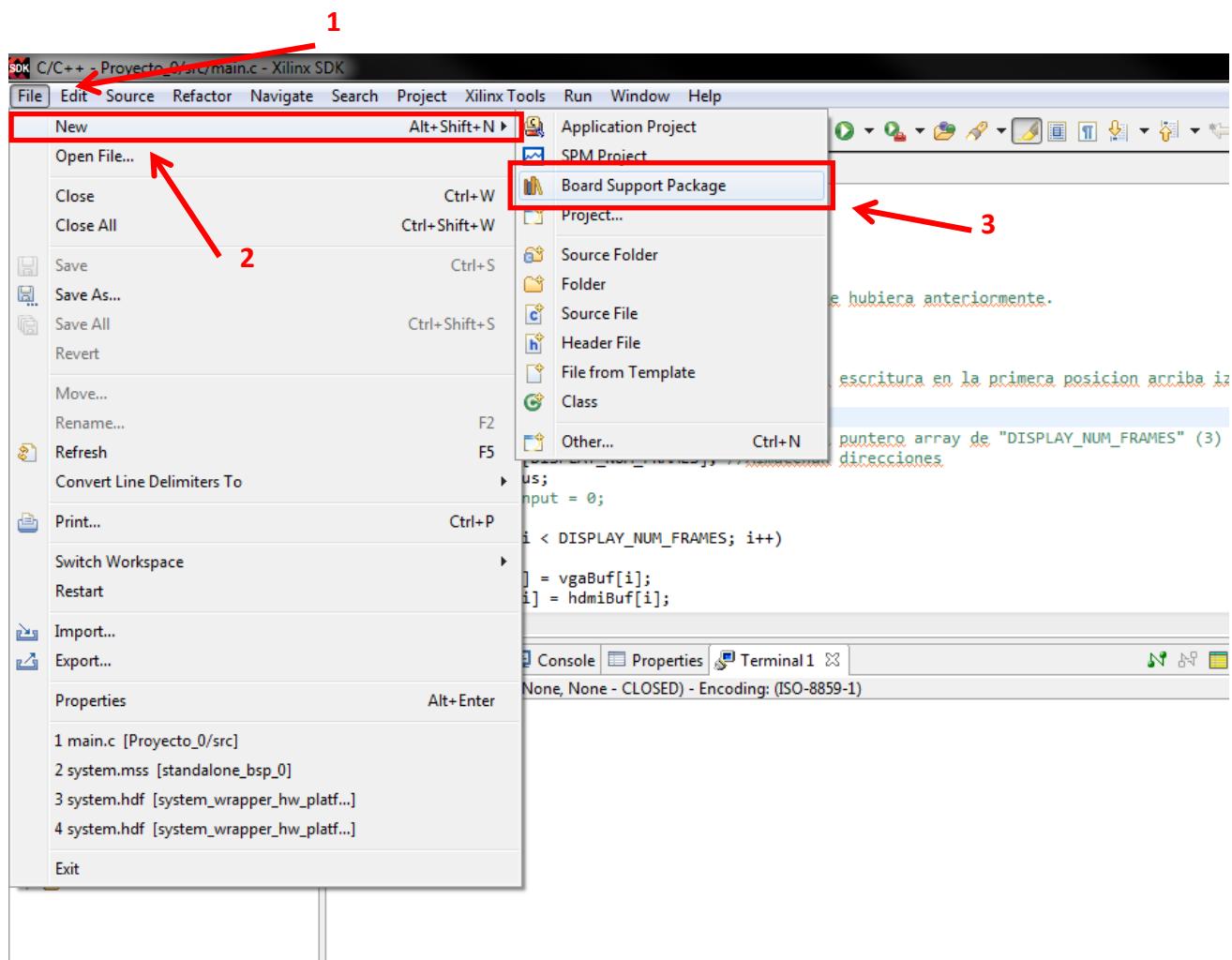


Ilustración 57. Crear un BSP (1)

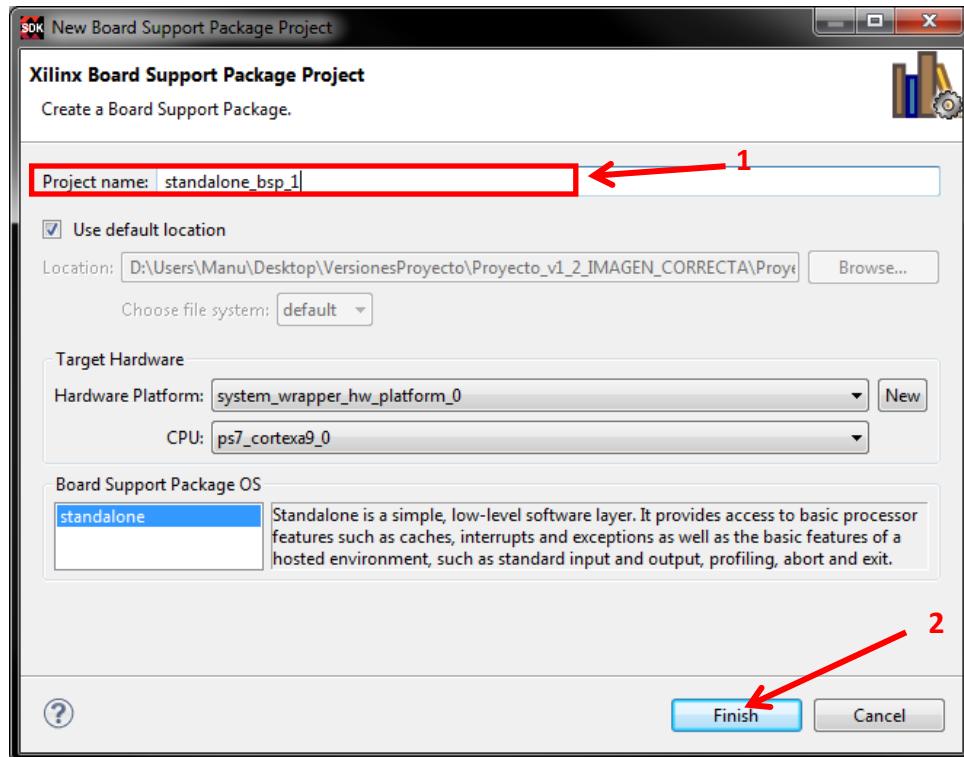


Ilustración 58. Crear un BSP (2)

Una vez que tenemos creado el BSP, podemos ver que se nos ha abierto en la pantalla principal del SDK un fichero "System.mss" el cual, podemos importar ejemplos del hardware añadido en nuestro diseño así como consultar su documentación:

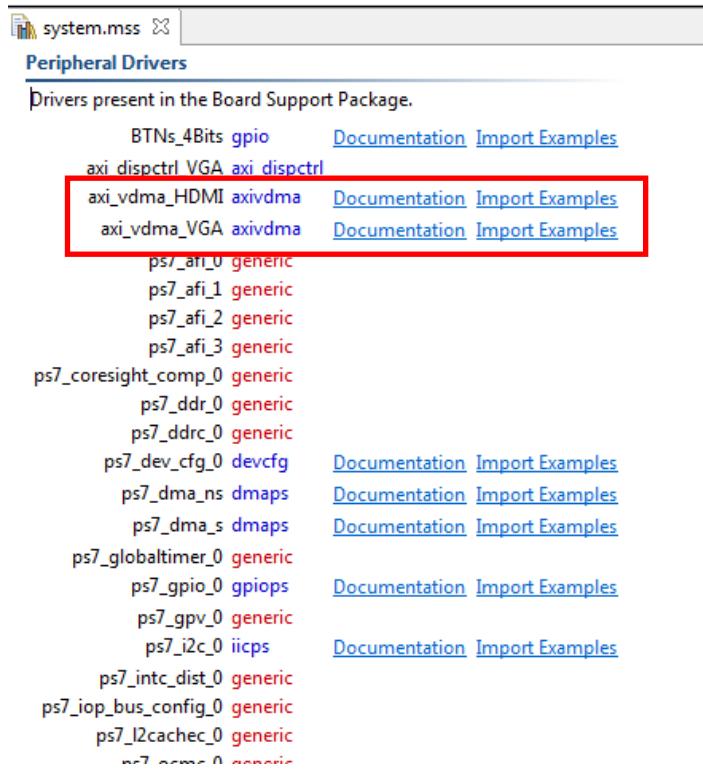


Ilustración 59. System.mss del BSP generado

Siguiendo la documentación, la configuración del VDMA se divide en 2 partes:

### **1. Inicializar el driver**

Para realizar esta inicialización hay que realizar una llamada a 3 funciones propias del VDMA que son:

- a) **XAxiVdma\_LookupConfig()** → Encargada de obtener la estructura de configuración del VDMA cuyo ID se especifique al realizar la llamada a la función.
- b) **XAxiVdma\_CfgInitialize()** → Encargada de iniciar el driver y el dispositivo, utilizando para ello la estructura obtenida con la función a) y un puntero a una estructura de tipo XAxiVdma.
- c) **XAxiVdma\_SetFrmStore()** → Utilizada para designar el número de fotogramas deseados. Aunque la llamada a esta función es opcional, en mi caso se realiza definiendo 3 frames (Pues es la configuración indicada en todos los VDMAs en el diseño realizado en Vivado).

### **2. Comenzar el intercambio**

Para esta tarea se pueden invocar las funciones “XAxiVdma\_StartWriteFrame” o “XAxiVdma\_StartReadFrame” o bien, realizar llamadas a funciones en un orden determinado. En el desarrollo de este proyecto, se utiliza ésta última:

- a) **XAxiVdma\_DmaConfig()** → Para realizar la configuración de la operación que realizará el DMA. Antes de esto, hay que crear una estructura del tipo **XAxiVdma\_DmaSetup** cuya configuración permite definir parámetros como: número de píxeles verticales/horizontales, el “stride” (o paso) entre un pixel y otro y el más importante, el “*FrameStoreStartAddr*”.

Este valor, indica la dirección de memoria desde donde el VDMA comenzará su intercambio de datos.

- b) **XAxiVdma\_DmaSetBufferAddr()** → Con esta función se configuran los buffers del DMA, y es aquí donde se indica al VDMA la dirección desde donde comenzar su tarea, el mencionado en a) “*FrameStoreStartAddr*” además de la dirección de trabajo: escritura o lectura.
- c) **XAxiVdma\_DmaStart()** → Para inicializar las operaciones que debe realizar el VDMA. Una vez que está todo configurado, indicamos al VDMA que ya puede comenzar.

### **3. Interrupciones**

En la documentación indica también cómo se deben controlar las interrupciones en los VDMAs, pero no es el caso que nos ocupa, pues en este proyecto no se hace uso de ellas.

#### 4. Configuración realizada en el diseño

Una vez comprendidos los pasos a realizar para configurar los VDMAs, ésta es la forma en la que hay que realizar la programación en el SDK para configurarlos y la configuración elegida en el core desde Vivado.

##### VDMA HDMI (Entrada de datos, escritura en memoria)

```

int iniciaVdmaHdmi2(int ID_VDMA, XAxiVdma_Config *CfgVdmaHdmi, XAxiVdma VdmaHdmi, XAxiVdma_DmaSetup
                      StpVdmaHdmi, u32 *puntero[], DisplayCtrl vgaCtrl){
    int Status,i;

    CfgVdmaHdmi = XAxiVdma_LookupConfig(ID_VDMA);
    Status = XAxiVdma_CfgInitialize(&VdmaHdmi, CfgVdmaHdmi, CfgVdmaHdmi->BaseAddress);

    if (Status == XST_SUCCESS){
        Status = XAxiVdma_SetFrmStore(&VdmaHdmi, 3, XAXIVDMA_WRITE);
    }else{
        return Status;
    }

    //Configuro los parametros necesarios para el VDMA
    StpVdmaHdmi.VertSizeInput = SUBFRAME_VERTICAL_SIZE;
    StpVdmaHdmi.HoriSizeInput = SUBFRAME_HORIZONTAL_SIZE;
    StpVdmaHdmi.Stride = FRAME_HORIZONTAL_LEN;
    StpVdmaHdmi.FrameDelay = 0; /* This example does not test frame delay */
    StpVdmaHdmi.EnableCircularBuf = 1;
    StpVdmaHdmi.EnableSync = 0; /* Gen-Lock */
    StpVdmaHdmi.PointNum = 1; /* No Gen-Lock */
    StpVdmaHdmi.EnableFrameCounter = 0; /* Endless transfers */

    for (i = 0; i < DISPLAY_NUM_FRAMES; i++){
        StpVdmaHdmi.FrameStoreStartAddr[i] = (u32) puntero[i];
    }

    Status = XAxiVdma_DmaConfig(&VdmaHdmi, XAXIVDMA_WRITE, &StpVdmaHdmi);
    Status = XAxiVdma_DmaSetBufferAddr(&VdmaHdmi, XAXIVDMA_WRITE, StpVdmaHdmi.FrameStoreStartAddr);
    Status = XAxiVdma_DmaStart(&VdmaHdmi, XAXIVDMA_WRITE);
    Status = XAxiVdma_StartParking(&VdmaHdmi, vgaCtrl.curFrame, XAXIVDMA_WRITE);

    if (Status == XST_SUCCESS){
        return Status;
    }else{
        xil_printf("Error al asignar la dirección del buffer. Status vale: %d\n\r",Status);
    }
}

```

Código 11. Configuración del VDMA HDMI

Se comienza, guardando en la variable CfgVdmaHdmi la configuración que la función XAxiVdma\_LookupConfig genera para el dispositivo con ID: ID\_VDMA el cual, en este caso, se pasa como argumento y es el ID que se asigna al VDMA que estamos configurando (El HDMI).

Con esa configuración obtenida (estructura tipo XAxiVdma\_Config), se inicializa el driver de control del dispositivo, indicando la instancia del VDMA en cuestión (&VdmaHdmi, de tipo XAxi\_Vdma), la configuración anterior y la dirección base de esa estructura. Si todo ha ido bien, se asignan 3 buffers de almacenamiento en el VDMA, indicando que será de tipo escritura, usando la variable XAXIVDMA\_WRITE en la función XAxiVdma\_SetFrmStore.

Adicionalmente, se configuran algunos parámetros necesarios en la estructura StpVdmaHdmi, de tipo XAxiVdma\_DmaSetup, donde se le indican:

- Píxeles verticales: 720.
- Píxeles horizontales: 1280.
- Tamaño del salto: desde un pixel, al siguiente: 4x1280, (ya que son R,G,B,Alfa).
- Retardo del frame: en esta configuración no se permite, por eso 0.
- Habilitar frame circular: Activo, por eso a 1.
- Habilitar sincronización: No es necesario, por eso a 0.
- “PointNum”: Se coloca a 1 para que coincida con la sincronización.
- Habilitar contador de frames: Se deja en 0 ya que habrá transferencia sin límite.

Para terminar, hay que establecer el FrameStoreStartAddr que indica desde qué posición de memoria empezará a trabajar el VDMA, en este caso, la dirección de memoria desde donde comenzará a escribir. Para esto, se realiza un bucle que copiará las direcciones de memoria en un puntero creado anteriormente por tener unas variables mejor indicadas, es decir, nombradas con nombres más significativos.

En el código anterior, se utiliza una variable llamada “Puntero” que se pasa como argumento y en el código main, se nombra esta variable como “HdmiPtr” con el objetivo de que sea más sencillo comprender el código.

Una vez que todo está configurado, se aplica esta configuración usando para ello la función XAxiVdma\_DmaConfig y se asigna la dirección desde donde comenzará a trabajar con XAxiVdma\_DmaSetBufferAddr.

Con todo el VDMA configurado y listo, se inicia el funcionamiento a través de la función XAxiVdma\_DmaStart y finalmente, se llama a la función XAxiVdma\_StartParking para que el vídeo que maneje el VDMA, se sincronice con el frame actual de la estructura “vgaCtrl” que es el core Display Controller encargado de mostrar la imagen por pantalla.

Una vez que se ha realizado todo este procedimiento, el VDMA HDMI queda configurado y funcionando, realizando escrituras en memoria en la dirección asignada anteriormente.

Para terminar, la configuración realizada del VDMA en Vivado, es la siguiente:



Ilustración 60. Configuración del VDMA HDMI en Vivado

Como puede observarse, sólo se activa el canal de escritura y se ha configurado de forma que el Stream Data Width sea de 32 bits. Éste tamaño es el que se espera en streams con color, y de otra forma, la imagen se obtendría perdiendo un canal de color y no llenaría completamente la pantalla de salida.

#### VDMA HLS (Lectura/escritura en memoria de streams de vídeo)

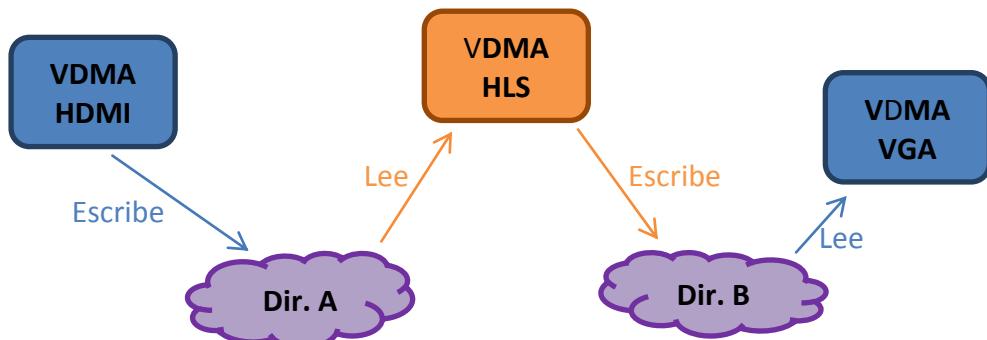
Este VDMA, se configura de una forma similar al descrito anteriormente, con la peculiaridad de que en este caso hay que configurar tanto la lectura como la escritura.

Para ello, se debe:

1. Asignar los 3 buffers de almacenamiento tanto para escritura como para lectura, utilizando la función XAxiVdma\_SetFrmStore con el parámetro XAXIVDMA\_WRITE y XAXIVDMA\_READ respectivamente, de forma similar a la explicada en el VDMA HDMI.
2. Se debe asignar al igual que para el VDMA HDMI, una dirección de memoria para la lectura y una dirección de memoria para la escritura.

En este caso, la dirección de memoria, debe ser la misma dirección de escritura del VDMA HDMI y la dirección de escritura, debe ser una dirección distinta que en el tercer VDMA se utilizará para leer.

En el siguiente esquema, se muestra de una forma sencilla el funcionamiento de los VDMAs utilizados en el proyecto. El VDMA de entrada (VDMA HDMI) escribe en la dirección A, el VDMA HLS lee de esa dirección, escribe en otra dirección B en la cual el VDMA de salida (VDMA VGA) leerá y mostrará por pantalla:



Esquema 4. Esquema diseño VDMAs

Para conseguir obtener una lectura y escritura de forma simultánea en el VDMA, hay que hacer primero la configuración de cada parte por separado. Por ejemplo se debe configurar una estructura XAxiVdma\_DmaSetup para la escritura (en mi código, StpWrite), a continuación se utiliza un bucle para asignar las direcciones donde escribirá el VDMA, se aplica la configuración con XAxiVdma\_DmaConfig indicando que será de escritura con la variable XAXIVDMA\_WRITE y se inicia la parte de escritura con XAxiVdma\_DmaStart de nuevo indicando que es escritura con la variable XAXIVDMA\_WRITE.

A continuación, se debe realizar el mismo procedimiento para configurar la parte de lectura, en mi caso utilizo StpRead como estructura de configuración, y en todo caso se debe usar XAXIVDMA\_READ para configurar la parte de lectura.

Después de toda la configuración de cada una de las partes y habiendo iniciado ambas con la función XAxiVdma\_DmaStart, también se llama a la función XAxiVdma\_StartParking para que los fotogramas vayan sincronizados al igual que en VDMA HDMI.

El código utilizado para la configuración de este VDMA es el siguiente:

```

int iniciaVdmaHls(int ID_VDMA, XAxiVdma_Config *CfgVdmaHls, XAxiVdma_VdmaHls, XAxiVdma_DmaSetup StpRead,
XAxiVdma_DmaSetup StpWrite, u32 *punteroRead[], u32 *punteroWrite[], DisplayCtrl vgaCtrl){
    int Status,i;

    //----- CONFIGURACION DE LA ESCRITURA -----\\
    CfgVdmaHls = XAxiVdma_LookupConfig(ID_VDMA);
    Status = XAxiVdma_CfgInitialize(&VdmaHls, CfgVdmaHls, CfgVdmaHls->BaseAddress);

    if (Status == XST_SUCCESS){
        Status = XAxiVdma_SetFrmStore(&VdmaHls, 3, XAXIVDMA_WRITE);
        Status = XAxiVdma_SetFrmStore(&VdmaHls, 3, XAXIVDMA_READ);
    }else{
        return Status;
    }

    StpWrite.VertSizeInput = SUBFRAME_VERTICAL_SIZE;
    StpWrite.HoriSizeInput = SUBFRAME_HORIZONTAL_SIZE;
    StpWrite.Stride = FRAME_HORIZONTAL_LEN;
    StpWrite.FrameDelay = 0; /* This example does not test frame delay */
    StpWrite.EnableCircularBuf = 1;
    StpWrite.EnableSync = 0; /* Gen-Lock */
    StpWrite.PointNum = 1; /* No Gen-Lock */
    StpWrite.EnableFrameCounter = 0; /* Endless transfers */

    for (i = 0; i < DISPLAY_NUM_FRAMES; i++){
        StpWrite.FrameStoreStartAddr[i] = (u32) punteroWrite[i];
    }

    Status = XAxiVdma_DmaConfig(&VdmaHls, XAXIVDMA_WRITE, &StpWrite);
    Status = XAxiVdma_DmaSetBufferAddr(&VdmaHls, XAXIVDMA_WRITE, StpWrite.FrameStoreStartAddr);
    Status = XAxiVdma_DmaStart(&VdmaHls, XAXIVDMA_WRITE);
    Status = XAxiVdma_StartParking(&VdmaHls, vgaCtrl.curFrame, XAXIVDMA_WRITE);

    //----- CONFIGURACION DE LA LECTURA -----\\

    StpRead.VertSizeInput = SUBFRAME_VERTICAL_SIZE;
    StpRead.HoriSizeInput = SUBFRAME_HORIZONTAL_SIZE;
    StpRead.Stride = FRAME_HORIZONTAL_LEN;
    StpRead.FrameDelay = 0; /* This example does not test frame delay */
    StpRead.EnableCircularBuf = 1;
    StpRead.EnableSync = 0; /* Gen-Lock */
    StpRead.PointNum = 1; /* No Gen-Lock */
    StpRead.EnableFrameCounter = 0; /* Endless transfers */

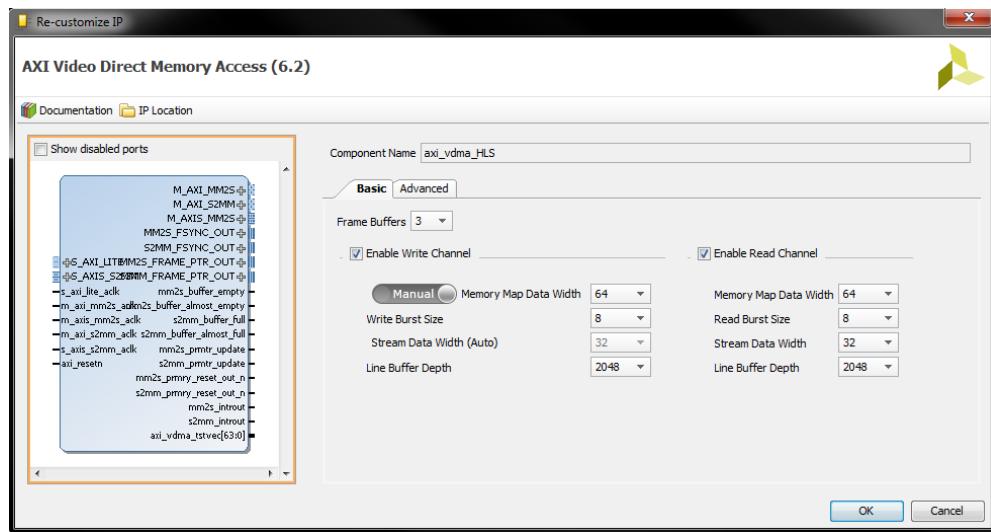
    for (i = 0; i < DISPLAY_NUM_FRAMES; i++){
        StpRead.FrameStoreStartAddr[i] = (u32) punteroRead[i];
    }

    Status = XAxiVdma_DmaConfig(&VdmaHls, XAXIVDMA_READ, &StpRead);
    Status = XAxiVdma_DmaSetBufferAddr(&VdmaHls, XAXIVDMA_READ, StpRead.FrameStoreStartAddr);
    Status = XAxiVdma_DmaStart(&VdmaHls, XAXIVDMA_READ);
    Status = XAxiVdma_StartParking(&VdmaHls, vgaCtrl.curFrame, XAXIVDMA_READ);
}

```

Código 12. Configuración del VDMA HLS (Lectura y escritura)

En cuanto a la configuración del VDMA en Vivado, es la siguiente:



**Ilustración 61. Configuración del VDMA HLS en Vivado**

Como se puede apreciar, en este caso están activas tanto la casilla de escritura, como la de lectura, pero ambas tienen la misma configuración entre sí, y a su vez, la misma configuración que con el VDMA HDMI.

### VDMA VGA (Lectura de streams de vídeo desde memoria)

Toda la descripción que se está detallando en estos apartados, se ha considerado interesante de cara a que el proyecto sea reproducible, ayudando a futuros desarrolladores a realizar el mismo diseño aquí presentado y comprobar la funcionalidad aquí descrita.

Este VDMA es más sencillo que el anterior. Se configura de una forma muy similar al VDMA HDMI, pero utilizando siempre el parámetro que indica lectura: XAXIVDMA\_READ.

Además, debe tenerse en cuenta a la hora de realizar la configuración, que la dirección a la que debe apuntar este VDMA para leer desde memoria, debe ser la misma que se utiliza para escribir en el VDMA HLS.

La configuración utilizada en Vivado para este VDMA es la siguiente:

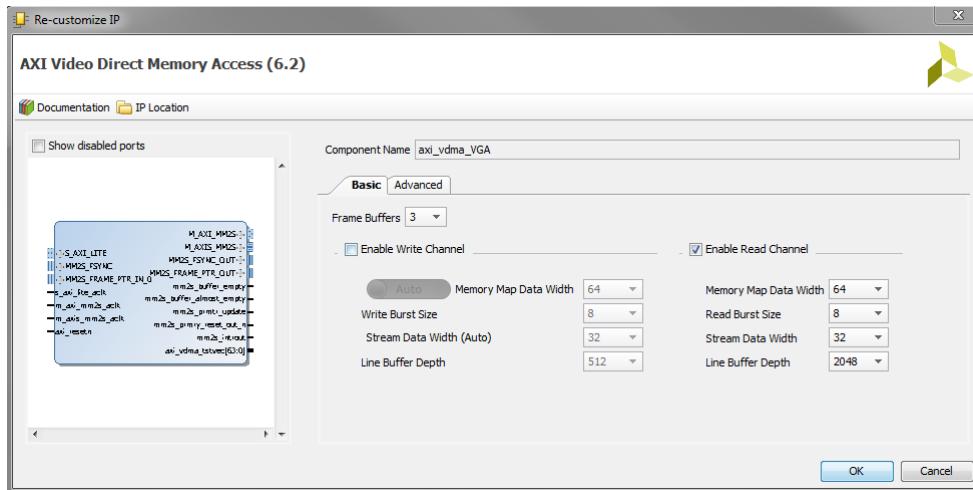


Ilustración 62. Configuración del VDMA VGA en Vivado

En este VDMA, está sólo activa la parte de lectura, pero de nuevo, tiene la misma configuración que los otros VDMAs para que esté todo coordinado y no haya problemas.

La programación de este VDMA en este caso, es más compleja en el SDK ya que, se va configurando a la vez que se configura el core encargado de mostrar la imagen por pantalla, el core Axi Display Controller, pero la programación es similar a la ya explicada anteriormente.

### 3.3.3 Consumo de recursos y compromiso de diseño

Una vez que todo el diseño está realizado en Vivado, se ha terminado la sintetización e implementación del mismo, terminando con la generación del Bitstream que programará la placa de desarrollo con la arquitectura creada, Vivado ofrece también una serie de datos de consumo de recursos de la placa y de condiciones de tiempo.

[FOR,15,A] Observando los datos de temporización del diseño, se muestran los valores del “Worst Negative Slack” y “Total Negative Slack”:

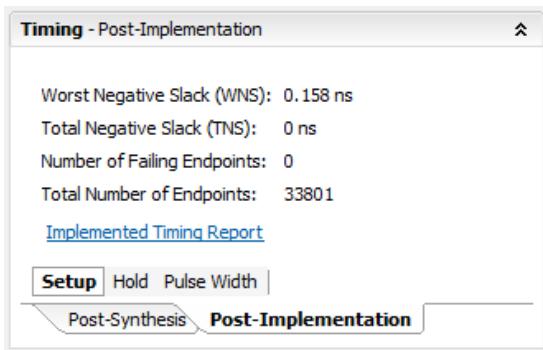


Ilustración 63. Datos de temporización (1)

El Slack que se asocia con cada señal, es la diferencia entre el tiempo requerido, que se establece en el fichero de requisitos (constraints) y el tiempo de llegada efectivo de la señal. Así, el Worst Negative Slack (WNS) indicaría un camino de datos demasiado lento, siendo necesaria por tanto una optimización del diseño para acelerarlo en el caso de precisar la velocidad indicada en el fichero de limitaciones pues, es posible que sea más importante la utilización de la placa, que el tiempo de ejecución.

Si este valor es positivo, significa que el diseño cumple con los requisitos de tiempo y que no es necesaria esa optimización para acelerar el diseño.

En definitiva, el WNS es el slack más alto en valor absoluto, de los slacks de un camino que hace que falle una limitación determinada.

A su vez, el Total Negative Slack (TNS) es la suma de todos los slacks negativos para las diferentes señales del diseño que hacen que fallen limitaciones de tiempo.

Siendo este valor 0, quiere decir que se cumplen sin problema todas las limitaciones de tiempo.

Por otro lado, está el Worst Hold Slack (WHS) y el Total Hold Slack (THS):

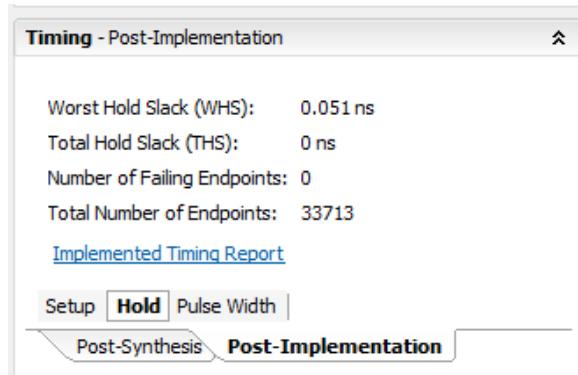


Ilustración 64. Datos de temporización (2)

Estas señales tienen un comportamiento análogo a las anteriores, donde el WHS es el hold slack (tiempo que debe mantenerse el valor de reloj para que no se produzca el slack) más alto que se produce en el diseño y el THS es la suma de los mismos.

De nuevo, cuando el THS es 0, indica que se cumplen todas las limitaciones de tiempo establecidas en el fichero de las mismas.

Y finalmente, el Worst Pulse Width Slack (WPWS) y el Total Pulse Width Negative Slack (TPWS):

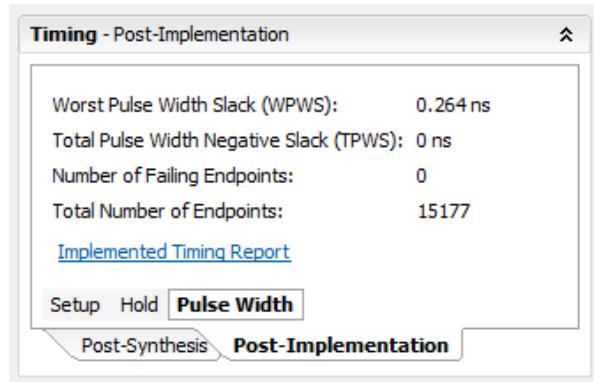


Ilustración 65. Datos de temporización (3)

El WPWS está referido al ancho mínimo de los pulsos en valor alto o valor bajo de una señal de reloj de manera que, de nuevo, no se den situaciones en las que no se cumplan las limitaciones de tiempo establecidas.

De una forma similar a las anteriores, el total (TPWS) siendo 0, indica que se han cumplido satisfactoriamente todas las limitaciones de tiempo requeridas por el diseño y que no habrá problemas por ello.

Vivado, también ofrece un resumen de la utilización de recursos de la placa, en la que se muestra tanto en formato gráfico como en formato tabla, el uso de cada uno de los dispositivos disponibles en la Zybo:

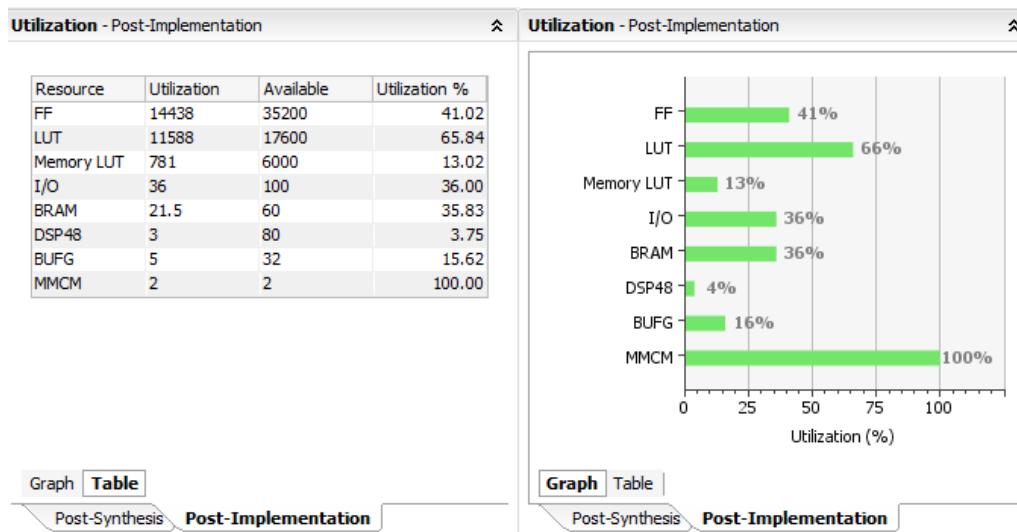


Ilustración 66. Utilización de recursos del diseño

Donde pueden verse que por ejemplo, se utilizan los 2 MMCM (Mixed-Mode Clock Manager) que forman parte de la FPGA de la Zybo, y que se utilizan para generar múltiples relojes definiendo la fase y la frecuencia en función de otras entradas de reloj. Sin embargo quedan aún recursos suficientes para añadir nuevos cores y ampliar las funcionalidades de la arquitectura, pues se observa que los LUTs, BlockRAMs y DSPs tienen aún bastante porcentaje libre.

Por último, otro de los datos interesantes que ofrece Vivado, es el consumo en potencia de la placa en función a la arquitectura diseñada, y son unos datos muy curiosos, pues:

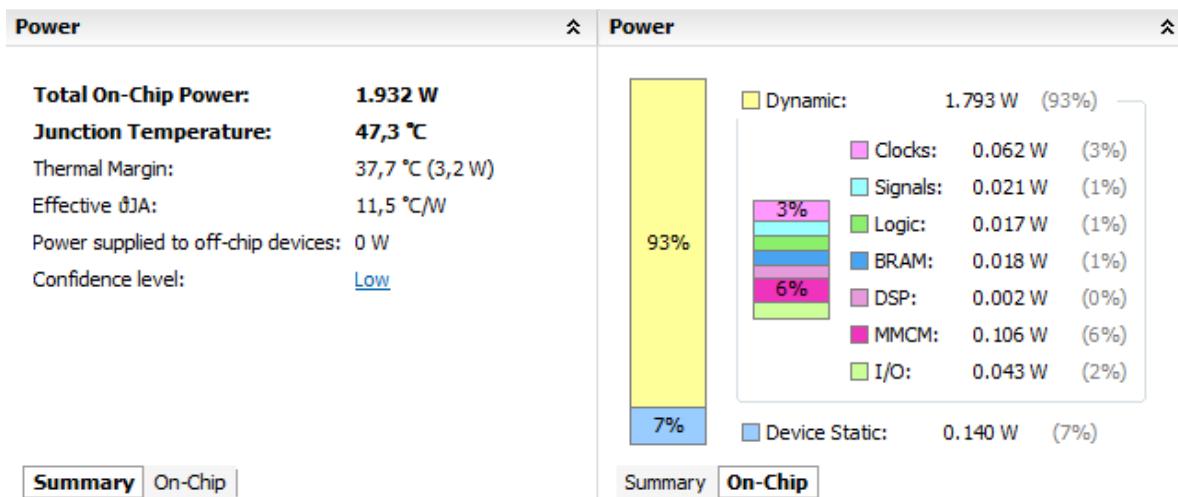


Ilustración 67. Potencia de consumo del diseño

Comprobando estos datos, vemos que el diseño en la placa consumirá unos 2W de pico (1.932 W) a una temperatura de 47.3 °C. Con estos datos, podemos ver que alimentar este diseño en la Zybo, podría realizarse con unas baterías sencillas pudiendo prolongarse durante suficiente tiempo el funcionamiento, si se utilizaran interrupciones que despertaran al procesador cuando fuera necesario.

## 4. Conclusiones.

Centrándonos en el proyecto en cuestión, creo que se ha demostrado claramente la potencia de los System on Chip con FPGA, pues se consigue con una placa de muy bajo consumo (puede alimentarse con 5V de un USB) un procesado de vídeo profesional con una latencia muy baja y procesando imágenes de alta resolución.

Cabe destacar también, que esta placa de desarrollo, es barata y ofrece gran variedad de interfaces: procesador de doble núcleo, FPGA, JTAG por microUSB... y permite la instalación de un Linux (Xilinx) funcional (aunque no muy optimizado) o bien, la ejecución de aplicaciones stand-alone (sin sistema operativo) permitiendo acceso completo a todos los periféricos sin uso de drivers.

Aunque no ha podido medirse de forma concisa los tiempos de ejecución y fotogramas por segundo de la imagen, es claro a la vista que la imagen no sufre latencia apreciable con la fuente del vídeo. Probando vídeo de alta resolución y haciendo pruebas de aceleración del mismo, se ve que no es problema ninguno para la placa, y el vídeo, se procesa y se muestra con una latencia casi inexistente.

Como opinión personal, las herramientas de Xilinx ofrecen grandes ayudas al desarrollador a la hora de realizar los diseños de las arquitecturas, pues incorporan

útiles herramientas de automatización de los diseños, permitiendo que en poco tiempo se pueda generar una arquitectura compleja.

Sin embargo, hay que dedicar bastante tiempo para conseguir entender todos los pasos a realizar para conseguir que los IP Cores funcionen correctamente y de la forma deseada y fundamentalmente a base de “prueba y error”. Siendo una herramienta que sintetiza consiguiendo una enorme eficiencia y ahorro (tanto en tiempos como en espacio en placa) necesita bastante tiempo para realizar toda la implementación necesaria hasta que se consigue el bitstream que se descarga en la placa para reorganizar las puertas de la FPGA con nuestro diseño.

No obstante, con Vivado, una vez que se han realizado los suficientes ciclos de aprendizaje del software, resulta muy cómoda la programación con bloques configurables a golpe de ratón, permitiendo al usuario, de una forma muy visual, la configuración de los cores haciéndolo sencillo.

Además, Xilinx SDK es una interfaz basada en eclipse, familiar para los programadores software, con una interfaz gráfica muy usada y extendida en el mundo de la programación, que ofrece gran comodidad y alta usabilidad. A parte de las herramientas propias de Xilinx, necesarias para interactuar con la placa de desarrollo, todo es muy similar a la base Eclipse.

También incluye una terminal desde la que podemos interactuar con la placa de desarrollo, permitiendo la comunicación tanto de los mensajes que envíe, como enviando mensajes, lo que puede ser muy útil a la hora de testear y verificar funcionamientos, además de que se ahorra memoria en el equipo de desarrollo pues no hay que abrir otro software (Como Hyperterminal o similares) para poder contar con una terminal.

Por último, Vivado HLS es otra herramienta muy útil. Es una interfaz, de nuevo basada en eclipse, muy sencilla de utilizar y que permite, programando en código C o C++, el lenguaje más extendido dentro de la programación. Permite también añadir directivas al código para conseguir aún más optimización del core diseñado y realizando, de forma automática, un core que podemos añadir sin problemas en Vivado, con las interfaces necesarias para realizar su conexión con el resto de cores de una manera sencilla. Incluyendo las restricciones que Vivado ofrece, pues si intentamos, por ejemplo, conectar una señal de tipo AXI Stream en una conexión para una señal binaria (un bit) no lo permite.

La única forma de conseguir un código más eficiente, es realizando programación a más bajo nivel, lo que es mucho más complejo.

Pensando en posibles aplicaciones que podrían darse a este desarrollo, se me ocurren bastantes, pues utilizando la arquitectura en la que ya se tiene configurado todo independientemente: entrada, salida y procesado usando VDMAs diferentes.

Si hablamos de utilizar el mismo proyecto, modificando el código del core de procesado, se podrían conseguir aplicaciones de seguimiento, background (eliminación de objetos que no se muevan), videovigilancia, clasificación de objetos/alimentos, etc...

Como trabajo futuro, se pueden conseguir aplicar diversos procesados a la imagen capturada, en función del valor indicado con los switches de la placa y dependiendo de la memoria disponible de la placa.

También se podría aplicar el modelo de background a la imagen, y eliminar de la imagen capturada todo aquello que no se mueva.

Tras todo esto, no me queda más que comentar el amplio aprendizaje conseguido con este proyecto, pues hasta ahora, no había trabajado en profundidad en la programación hardware, y es así donde te das cuenta de todo el trabajo que tiene la realización de sistemas específicos.

Después de más de un año de trabajo, creo que todo lo aprendido durante este duro desarrollo, se suma a mi formación dentro de la ingeniería en telecomunicaciones, que más de uno puede pensar que se queda en lo puramente teórico y podría abrirme puertas en otros ámbitos de la ingeniería del hardware aplicada a las telecomunicaciones, o bien, dentro de equipos de desarrollo para sistemas más complejos.

Desde otro punto de vista, tras todo este trabajo, es cuando notas que el auto aprendizaje es, ha sido, y será siempre parte del mundo de la ingeniería, y aunque desde que empecé a estudiar esta carrera, sabía que ya no dejaría de estudiar para aprender cosas nuevas, adaptarme a nuevas tecnologías e incluso, quien sabe, desarrollar nuevos dispositivos para el mundo.

Y por supuesto, me encanta.

## 5. Bibliografía

---

[DIG, 15, A] Digilent Inc. "Zybo Zynq Features". *Zybo Zynq – 7000 development board*. <http://digilentinc.com/Products/Detail.cfm?NavPath=2,400,1198&Prod=ZYBO>

[DIG, 15, B] Digi56. *Vivado Logo*.  
[http://www.digi56.hol.es/images/4/45/Vivado\\_logo.jpg](http://www.digi56.hol.es/images/4/45/Vivado_logo.jpg)

[ENR, 15] Enrique Jaime Fernández Sánchez. "Modelos de visión para tareas de videovigilancia en sistemas empotrados". *PhD Thesis Dissertation*. Mayo 2015.

[FOR,15,A] Avrum Warshawsky. "Total Negative Slack vs Worst Negative Slack". *Xilinx forums*. <https://forums.xilinx.com/t5>Welcome-Join/Total-Negative-Slack-vs-Worst-Negative-Slack/m-p/308095#M3771>

[GIT,15] Andreas Traber. "Base System Design for ZYBO FPGA Platform". *GitHub Repository* [https://github.com/atraber/zybo\\_base](https://github.com/atraber/zybo_base)

[INS, 15] 'LariSan'. "Quick Start Test Demo: Zybo Zynq 7000 Image Filtering Demo + GoPro". *Instructables: Share what you make*.  
<http://www.instructables.com/id/Quick-Start-Test-Demo-Zybo-Xilinx-Zynq-7000-Image-F/?lang=es>

[JAV,10, A] Javier Díaz Alonso. "Arquitecturas de altas prestaciones para procesamiento de imágenes basadas en hardware reconfigurable". *Máster en Sistemas Electrónicos para Entornos Inteligentes. Diapositiva 10*.

[LOU,ROS,MAR,DAV,15] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, David Northcote. "The Zynq Book Tutorials for Zybo and Zedboard".  
<http://www.zynqbook.com/download-book.html>

[SHO,15] Shop Trenz Electronic. [http://shop.trenz-electronic.de/media/image/thumbnail/25398\\_0\\_720x600.jpg](http://shop.trenz-electronic.de/media/image/thumbnail/25398_0_720x600.jpg)

[SYN,15] Emmanuel García, "Sistemas Embebidos y Ejemplos".  
<http://synnick.blogspot.com.es/2012/02/sistemas-embebidos-y-ejemplos.html>

[WIK, 15, A] Xilinx. "OpenCV Installation on ARM/Linux/Windows". *Wiki Xilinx*  
<http://www.wiki.xilinx.com/OpenCV+Installation>

[WIK, 15, B] Xilinx. "Wiki of Vivado HLS Library"  
<http://www.wiki.xilinx.com/HLS+Vídeo+Library>

[WIK, 15, C] Wikipedia. "Circuito integrado de aplicación específica".  
[http://en.wikipedia.org/wiki/Application-specific\\_integrated\\_circuit](http://en.wikipedia.org/wiki/Application-specific_integrated_circuit)

[WIK, 15, D] Wikipedia. "Sistema Embebido"  
[https://es.wikipedia.org/wiki/Sistema\\_embebido](https://es.wikipedia.org/wiki/Sistema_embebido)

[WIK, 15, E] Wikipedia. "Unidad de procesamiento gráfico"  
[https://es.wikipedia.org/wiki/Unidad\\_de\\_procesamiento\\_gr%C3%A1fico](https://es.wikipedia.org/wiki/Unidad_de_procesamiento_gr%C3%A1fico)

[WIK,15,F] Wikipedia. “Procesador digital de señales”

[https://es.wikipedia.org/wiki/Procesador\\_digital\\_de\\_señales](https://es.wikipedia.org/wiki/Procesador_digital_de_señales)

[WIK, 15, G] Wikipedia. “Field Programmable Gate Array”.

[https://es.wikipedia.org/wiki/Field\\_Programmable\\_Gate\\_Array](https://es.wikipedia.org/wiki/Field_Programmable_Gate_Array)

[XAT, 15] Carlos Zahumenszky. “QuadHD, 4K y 2K. Todo lo que siempre quisiste saber sobre la futura super-alta definición” <http://www.xataka.com/alta-definicion/quadhd-4k-y-2k-todo-lo-que-siempre-quisiste-saber-sobre-la-futura-super-alta-definicion>

[XIL, 14,A] Xilinx. “LogiCORE IP Vídeo In To AXI4-STREAM v3.0”. *Xilinx IP Documentation*.

[http://www.xilinx.com/support/documentation/ip\\_documentation/v\\_vid\\_in\\_axi4s/v3\\_0/pg043\\_v\\_vid\\_in\\_axi4s.pdf](http://www.xilinx.com/support/documentation/ip_documentation/v_vid_in_axi4s/v3_0/pg043_v_vid_in_axi4s.pdf)

[XIL, 15,A] Stephen Neuendorffer, Thomas Li, Devin Wang. “Accelerating OpenCV applications with Zynq-7000 All Programmable SoC using

[http://www.xilinx.com/support/documentation/application\\_notes/xapp1167.pdf](http://www.xilinx.com/support/documentation/application_notes/xapp1167.pdf)

[XIL, 15, B], Xilinx. “AXI Vídeo Direct Memory Access v6.2”. *Xilinx IP Documentation*.

[http://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_vdma/v6\\_2/pg020\\_axi\\_vdma.pdf](http://www.xilinx.com/support/documentation/ip_documentation/axi_vdma/v6_2/pg020_axi_vdma.pdf)

[XIL,15,C], Xilinx, “High-Level Synthesis”. *Vivado Design Suite User Guide*.  
[http://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2014\\_1/ug902-vivado-high-level-synthesis.pdf](http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf)

[XIL,15,D], Xilinx, “AXI Reference Guide”. *Vivado Design Suite*.  
[http://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_ref\\_guide/latest/ug1037-vivado-axi-reference-guide.pdf](http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf)

[XIL,15,E], Xilinx, “AXI Interconnect v2.1”. *LogiCORE IP Product guide*.  
[http://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_interconnect/v2\\_1/pg059-axi-interconnect.pdf](http://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf)

[XIL,15,F] Xilinx, “AXI GPIO v2.0”. *LogiCORE IP Product Guide*.  
[http://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_gpio/v2\\_0/pg144-axi-gpio.pdf](http://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf)

[XIL,15,G] Xilinx, “Processing System 7 v5.5”. *LogiCORE IP Product Guide*.  
[http://www.xilinx.com/support/documentation/ip\\_documentation/processing\\_system7/v5\\_5/pg082-processing-system7.pdf](http://www.xilinx.com/support/documentation/ip_documentation/processing_system7/v5_5/pg082-processing-system7.pdf)

[ZAT, 15] Zator Systems. “Percepción humana del color”. *Física del color*.  
[http://www.zator.com/Hardware/H9\\_1.htm](http://www.zator.com/Hardware/H9_1.htm)

## ANEXO A. Detalle de los distintos cores usados.

### 1. Core HDMI\_RX

Este core, es el dedicado a realizar la captura del vídeo en alta definición, a través del puerto HDMI que incluye la placa. Si vemos en detalle el core:

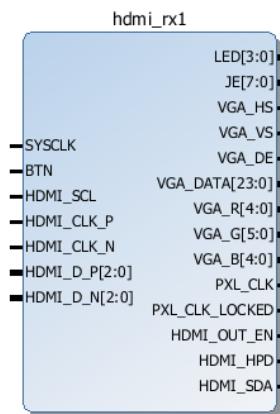


Ilustración 68. Detalle del core HDMI\_RX.

#### 1.1 Entradas

- **SYSCLK:** Señal de reloj. En mi diseño, se ha utilizado uno de los relojes del procesador, a una frecuencia de 200 MHz ya que esta es la frecuencia que se utilizaba en el diseño de referencia usado de la GoPro (Ilustración 5).
- **HDMI(SCL, CLK, D):**
  - Señales de reloj y de datos del vídeo en alta definición. Utilizadas para sincronización de datos y de actuación del core.

#### 1.2 Salidas:

- **JE:** Puerto para usar como debug.
- **VGA (HS, VS, DE):**
  - Señales de sincronización vertical y horizontal del vídeo, y señal de datos válidos.
- **VGA (DATA, R, G, B)**
  - Bits de datos del vídeo en VGA, tanto un bus con 24 bits de datos de la imagen, como cada canal por separado.
- **PXL\_CLK:** Reloj de pixel para indicar el reloj de los datos de imagen.
- **HDMI\_OUT\_EN:** Señal que indica la activación de salida del HDMI
- **HDMI\_HPD:** “Hot-Plug Detect” indica si hay una señal de vídeo. Es el mecanismo que utilizan los dispositivos para saber si hay algo conectado al HDMI o no.
- **HDMI\_SDA:** Línea de datos del HDMI.

## 2. Core AXI Display Controller

Se trata de un core propio de Digilent con el que, a través de un stream tipo AXI y una configuración por software a través del SDK, es posible obtener la salida del vídeo utilizando los pines propios de la VGA de la Zybo y las señales de sincronización vertical y horizontal de una manera relativamente sencilla.

Este core, se proporciona con el ejemplo de diseño ya mencionado “Zybo Base Design”:

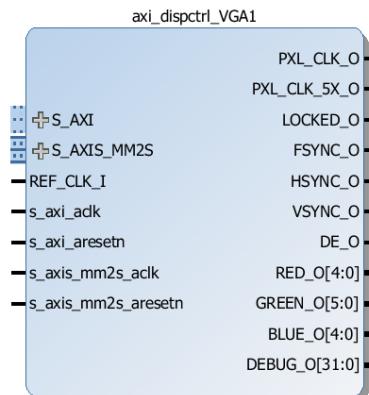


Ilustración 69. Salidas y entradas core Display Controller

### 2.1 Entradas:

- **S\_AXI:** Entrada de un stream tipo AXI, procedente del AXI Interconnect propio de Xilinx.
- **S\_AXIS\_MM2S:** Entrada tipo esclava de tipo “Memory Map To Stream” para lectura desde memoria y conversión a Stream. Señal procedente del VDMA.
- **REF\_CLK\_I:** Entrada para reloj de referencia. En este caso, se utiliza un reloj de 100 MHz al igual que en el diseño “Zybo Base Design” de ejemplo de Xilinx.
- **S\_axi\_aclk:** Entrada de reloj AXI. Se utiliza el mismo reloj que el de referencia (100 MHz) de la entrada anterior.
- **S\_axi\_aresetn:** Entrada de señal de reset para el AXI anterior.
- **S\_axis\_mm2s\_aclk:** Entrada de reloj para el stream AXI tipo “Memory Map To Stream”.
- **S\_axis\_mm2s\_aresetn:** Entrada con lógica negativa, para reset del stream AXI de tipo “Memory Map To Stream”

## 2.2 Salidas

- **PXL\_CLK\_O:** Reloj del pixel. Normalmente, es una señal de 25.175 MHz.
- **FSYNC\_O, HSYNC\_O, VSYNC\_O:** Señales de sincronización para la frecuencia, sincronización horizontal y vertical. Se utiliza como entrada al VDMA para la sincronización de la frecuencia del vídeo entre ambos módulos.
- **RED\_O:** 5 bits de salida para el color ROJO del VGA. Conectada directamente al pin correspondiente del puerto VGA de la Zybo.
- **GREEN\_O:** 6 bits de salida para el color VERDE del VGA. Conectada directamente al pin correspondiente del puerto VGA de la Zybo.
- **BLUE\_O:** 5 bits de salida para el color AZUL del VGA. Conectada directamente al pin correspondiente del puerto VGA de la Zybo.
- **DEBUG\_O:** Puerto para pruebas de depuración, desconectado.

## 3. Core Vídeo In to AXI4-Stream

[XIL, 14,A] Este core es el encargado de transformar un stream de datos de vídeo (bits continuos) en un stream de tipo AXI con el que se pueda trabajar con el resto de cores de Xilinx. Se utiliza como paso intermedio entre el vídeo obtenido a través del HDMI\_RX y el VDMA HDMI usado para la entrada, que se detallará más adelante. En este diseño se utiliza la versión v3.0 del core.



Ilustración 70. Entradas y salidas del core Vídeo In To Axi4-Stream

### 2.1 Entradas

- **Vid\_active\_video:** Indica bits de datos de vídeo activo, es decir, cuándo se está enviando datos de vídeo y no de sincronización.
- **Vid\_data:** Bits de datos de vídeo.
- **Vid\_field\_id:** Se utiliza para entrelazado de vídeo. En mi diseño, está anclado a un valor 0 para indicar que no se realiza entrelazado.
- **Vid\_(hblank, hsync, vblank, vsync):** Señales de sincronización de señal, tanto de datos de vídeo, como de datos “blancos” en horizontal y en

vertical. Sólo es necesario usar una pareja. En mi diseño, se utilizan hsync y vsync por ser proporcionados por el core HDMI\_RX.

- **Vid\_io\_in\_clk:** Reloj para la señal del vídeo de entrada.
- **Rst:** Señal de reset.
- **Vid\_io\_in\_ce:** Activación del reloj para la señal del vídeo de entrada. Puesto siempre a señal alta indicando que no se utiliza.
- **Aclk:** Reloj para el core.
- **Aclken, aresetn, axis\_enable:** Todas puestas a nivel alto. Necesarias para que el core se active y funcione correctamente.

## 2.2 Salidas

- **M\_axis\_vídeo\_tdata:** Señal de vídeo, ya transformada en un stream tipo AXI para poder ser manejada por los VDMAs.
- **M\_axis\_vídeo\_(tlast, tready, tuser, tvalid):** Señales de control de transferencia entre el VDMA y éste core. Se usan para saber cuándo comienza y acaba un stream, cuando son valores válidos... Para más información, [XIL, 14,A]

## 4. Core VDMA.

[XIL, 15, B] Estos cores, son los encargados de realizar todo el movimiento de datos, en este caso de vídeo, entre la memoria y los demás dispositivos del diseño y viceversa. Trabajan siempre con señales del tipo AXI Stream, y es por ello que es necesario el uso de cores como por ejemplo el mencionado en el punto 3 de este anexo A: Axi Vídeo In to AXI4Stream.

A su vez, será necesario el proceso inverso si se necesita trabajar con otro tipo de datos que no sean del tipo AXI. Para este diseño, se está usando la versión v6.2 del core.

En la arquitectura de este proyecto, se cuenta con 3 VDMAs distintos, donde:

1. El primer VDMA, es el encargado de escribir en memoria el stream de datos tipo AXI que recibe del core Vídeo In To AXI4Stream. Denominado en el diseño como “AXI\_vdma\_HDMI” pues es el encargado de manejar la señal que proviene del core HDMI\_RX.
2. El segundo VDMA, es el que se encarga de transferir los datos de vídeo, tanto lectura como escritura, entre la memoria de la tarjeta y el core HLS encargado de realizar el tratamiento (filtrado) de la imagen de vídeo. Se denomina este VDMA en el diseño como “AXI\_vdma\_HLS”.
3. El tercer y último VDMA, es el que se encarga de realizar la lectura del stream de vídeo desde la memoria de la placa para transferirlo por su salida al core AXI Display Controller que se encargará de transformar los datos tipo AXI a un stream de bits para poder representarlo en un dispositivo de salida. Este último VDMA se ha denominado en el diseño como AXI\_vdma\_VGA.

Para introducir de la mejor forma este core, se va a describir el VDMA comentado en el punto 2, pues éste se utiliza tanto para lectura como para escritura y por tanto es el que incluye más entradas/salidas. Éste es el core:

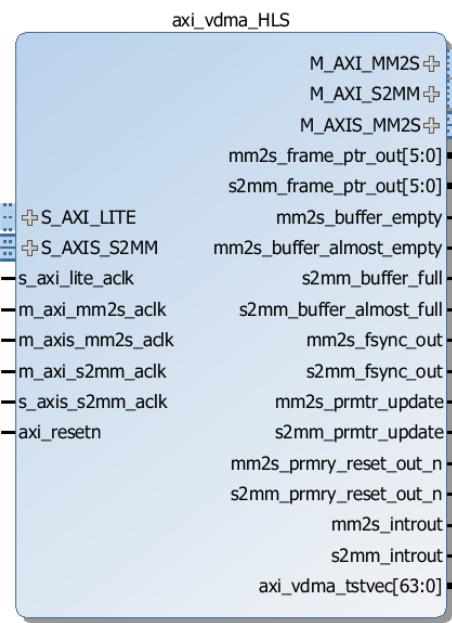


Ilustración 71. Entradas y salidas del core AXI Vídeo Direct Memory Access con lectura/escritura

#### 4.1 Entradas

- **S\_AXI\_LITE:** [XIL,15,D] Interfaz de control de señales tipo AXI Lite esclavas. Se conecta al core “AXI Interconnect” que se encarga de realizar las operaciones con los cores AXI. Todo core de tipo AXI debe conectarse así para poder usarse.
- **S\_AXIS\_S2MM:** Entrada de tipo “AXI Stream \_ Stream To Memory Map” es decir, es donde se debe conectar el stream de vídeo de tipo AXI que queramos escribir en la memoria. La primera S, indica que es una señal esclava (Slave). Si se despliega esta entrada con el símbolo ‘+’ que aparece a su lado, podemos conectar de forma individual las señales que deberían incluirse en el stream:



Ilustración 72. Entrada S\_AXIS\_S2MM en detalle.

Donde se encuentran:

1. Tdata: Señal de datos. Ésta debe ser la señal de vídeo, con 32 bits.

2. El resto de señales, son señales de control que indicarán al VDMA cuándo empieza/termina un stream de vídeo para que la transferencia de datos se haga de forma correcta. Deben proporcionarse o bien, unir las señales a nivel alto ó bajo (Según señal) para indicar que no se utilizarán.

- **S\_axi\_lite\_aclk:** Frecuencia de reloj que usará el core para las señales de tipo AXI Lite esclavas (Slave AXI Lite)
- **M\_axi\_mm2s\_aclk:** Frecuencia de reloj que usará el core para las señales de tipo AXI de lectura/escritura maestras (Master AXI memory map to stream y stream to memory map en la entrada: **m\_axi\_s2mm\_aclk**).
- **M\_axis\_mm2s\_aclk:** Frecuencia de reloj que usará el core para señales de tipo AXI Stream de lectura maestras (Master AXI Stream memory map to stream)
- **S\_axis\_s2mm\_acl:** Frecuencia de reloj que usará el core para señales de tipo AXI Stream de escritura (AXI Stream, Stream to memory map). En mi caso, la misma frecuencia que la señal anterior.
- **Axi\_resetn:** Señal de reset para reiniciar el core.

#### 4.2 Salidas

- **M\_AXI\_MM2S:** Interfaz de control de señales AXI Lite maestras de lectura (Master AXI memory map to stream). Se conecta al AXI Interconnect que se encargará de realizar las distintas operaciones con los AXI Lite.
- **M\_AXI\_S2MM:** Interfaz de control de señales AXI Lite maestras de escritura (Master AXI stream to memory map). Al igual que la anterior, debe conectarse al AXI Interconnect.
- **M\_AXIS\_MM2S:** Salida del stream de lectura maestra. Con esta señal se obtiene el stream de vídeo tipo AXI leído desde memoria: Master AXI Stream, memory map to stream. Al igual que ocurre con la señal de entrada, si desplegamos con el símbolo '+':

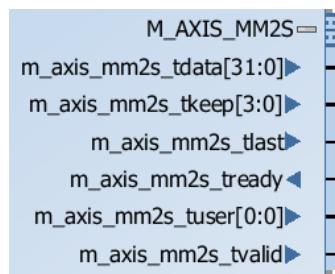


Ilustración 73. Salida M\_AXIS\_MM2S en detalle

Donde tenemos las mismas señales que en la entrada, pero con otra dirección. Tdata son los 32 bits de la señal de vídeo, y el resto de señales, son señales de control para indicar al VDMA cuándo debe hacer la transferencia de los datos para que se realice de forma sincronizada.

El resto de señales de salida, no las he utilizado, pero puede verse su uso en: [XIL, 15, B].

## 5. Core Constant

Éste es un sencillo core de Vivado, que como su propio nombre indica, fija un valor constante que se puede definir en su configuración. En mi diseño, se ha usado la versión v1.1.

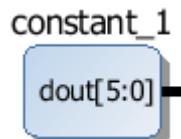


Ilustración 74. Detalle del core constant

Haciendo doble clic sobre el core, podemos realizar la configuración del mismo, la cual se reduce a:

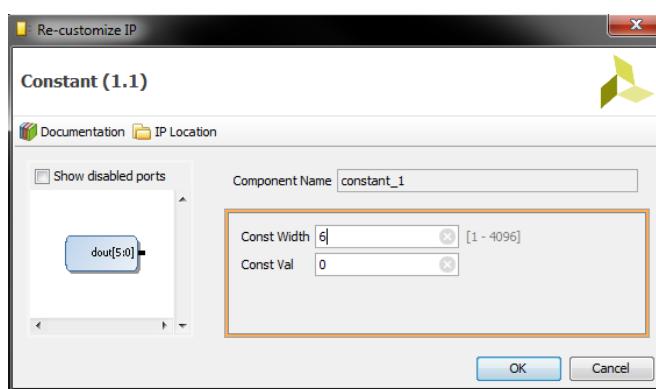


Ilustración 75. Ventana de configuración del core Constant

Podremos seleccionar el número de bits necesarios y el valor constante que proporcionará el core.

En mi diseño, lo he usado para tener una señal fija a 0 (Core LOW) una señal fija a 1 (Core HIGH) y otro, con valor “000000” usado como puntero para los frames de video en los VDMAs de entrada y salida al igual que en el diseño de referencia utilizado (Punto 4.1.2).

Una vez configurado, sólo hay que conectar la salida de este core, a la entrada donde necesitemos el valor fijo.

## 6. Core AXI Interconnect

<sup>[XIL,15,E]</sup> Este core, es el encargado de realizar la conexión entre las interfaces esclavas y maestras de los cores tipo AXI, de manera que puedan tener una interacción correcta entre ellos de una forma transparente para el desarrollador.

Con la automatización de conexiones de Vivado, este Core se crea automáticamente cuando se añade cualquier core tipo AXI al diseño.

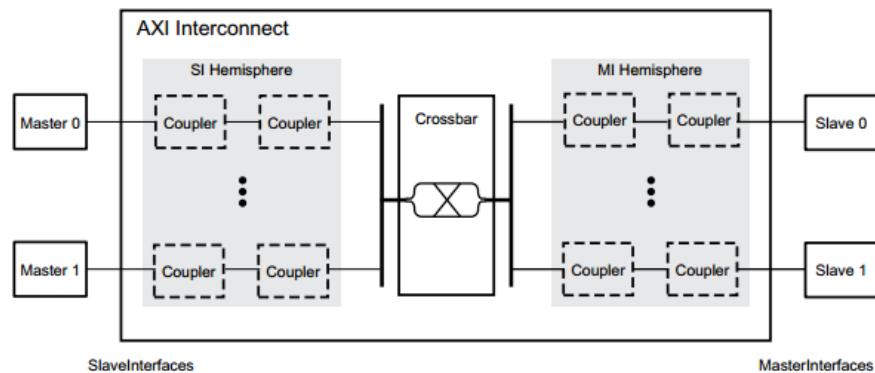


Ilustración 76. Diagrama de conexiones internas del AXI Interconnect [XIL,15,E] Figura 2-1

Pueden configurarse hasta 16 maestros y 16 esclavos sin tener que haber el mismo número de ambos. En el diseño aquí presentado, se han utilizado dos cores AXI Interconnect: uno del tipo 1-to-N para conectar el procesador de la ZYNQ como maestro (a través de un core AXI Protocol Converter) a todos los cores tipo AXI como esclavos, es decir, conexiones del procesador con los periféricos, y otro N-to-1 que conecta todos los cores AXI maestros al procesador como esclavo, para acceso a memoria.

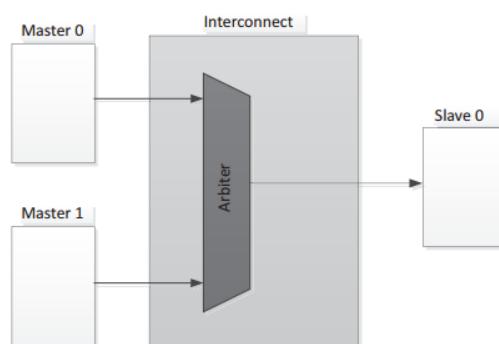


Ilustración 77. Interconexión realizada por el AXI Interconnect tipo N-to-1. [XIL,15,E] Figura 2-2

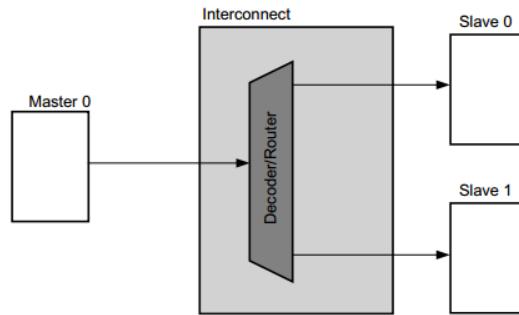


Ilustración 78. Interconexión realizada por el AXI Inerconnect tipo 1-to-N. [XIL,15,E] Figura 2-3

El core tipo N-to-1, está denominado en la arquitectura como: "axi\_mem\_intercon":

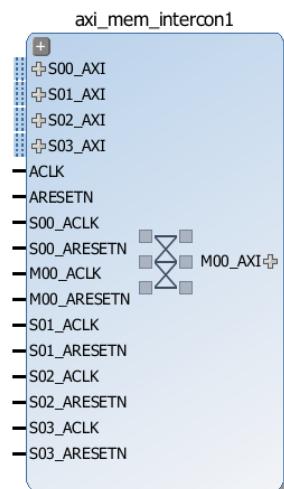


Ilustración 79. AXI Interconnect tipo N-to-1

Y el core tipo 1-to-N está nombrado "processing\_system7\_0\_axi\_periph"

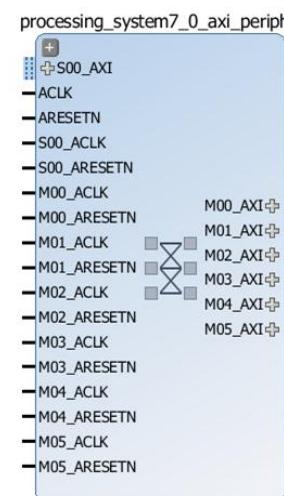


Ilustración 80. AXI Interconnect tipo 1-to-N

Además. Vivado muestra un curioso esquema de la conexión entre las entradas y las salidas, pulsando sobre el cuadro gris con una '+' en su interior:

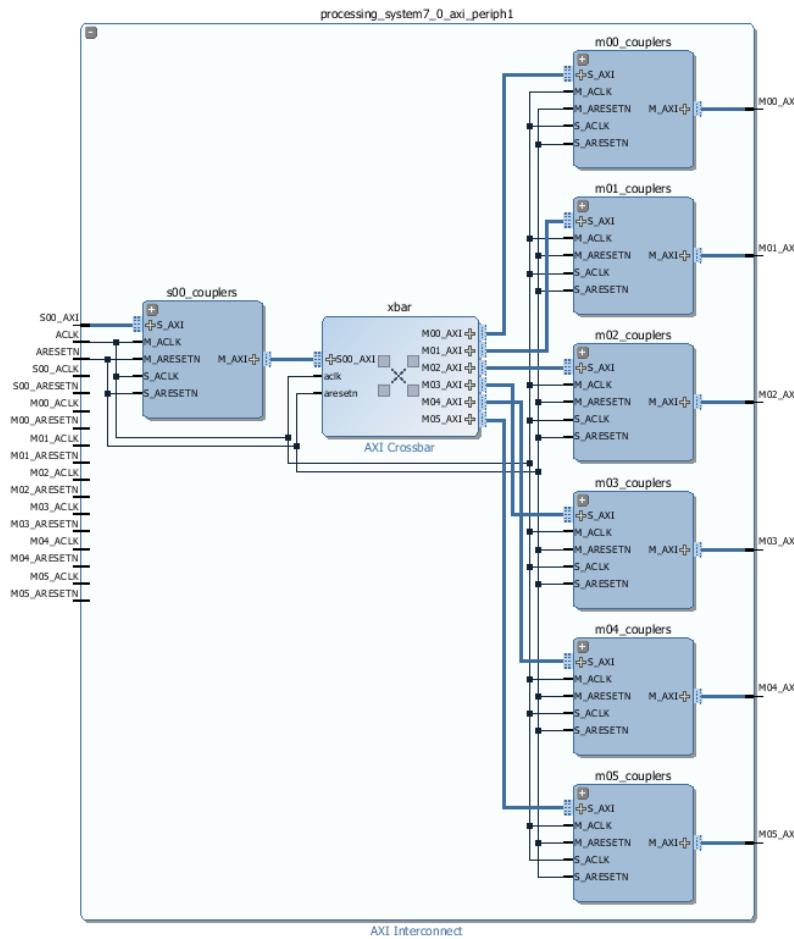


Ilustración 81. Detalle interior del AXI Interconnect 1-to-N

## 6.1 Entradas

En este core, tendremos como entradas los esclavos que se vayan a conectar, tanto sus señales de reloj, sus señales de reset y además, las señales de reloj y de reset de los maestros que estarán a la salida.

También, hay una señal de reloj y una señal de reset para el propio AXI Interconnect. En mi caso, el reloj de la señal ACLK (la del propio AXI Interconnect) es la misma para todos los esclavos y maestros y de la misma manera lo es la señal de reset.

## 6.2 Salidas

En este caso, las salidas serán las señales a cores maestros que se quieran conectar a la red AXI.

## 7 AXI Protocol Converter

<sup>[XIL,15,E]</sup> El Core AXI Protocol Converter, conecta un maestro de tipo AXI4, AXI3 o AXI4-Lite a un esclavo con un protocolo AXI diferente, de esta manera, la conversión de formato se realiza de forma transparente para el desarrollador, permitiendo aplicar distintas configuraciones. Para este proyecto, se utiliza la versión v2.1 del core, el cual se añade al proyecto de forma automática al realizar la conexión automatizada.

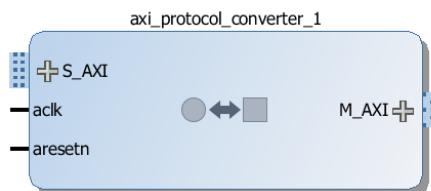


Ilustración 82. Core AXI Protocol Converter

Podemos acceder a la configuración del core, haciendo doble clic sobre el mismo:

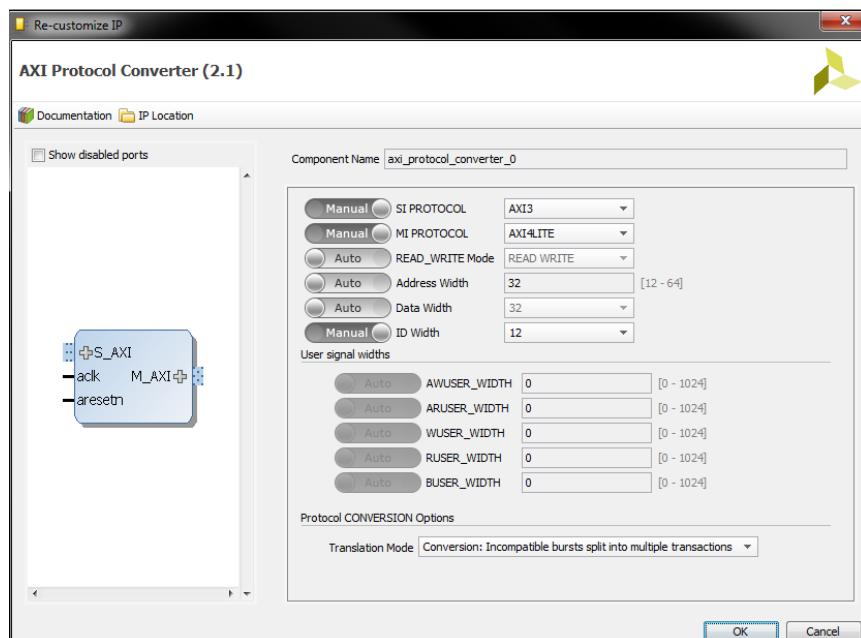


Ilustración 83. Configuración del core AXI Protocol Converter

Permitiendo desde la configuración, elegir el protocolo del maestro, del esclavo, y otros datos como el tipo de transacciones (lectura/escritura, lectura o escritura) el tamaño del bus de datos o el tamaño de las direcciones, etc...

### 7.1 Entradas

- **S\_AXI:** Señal esclava tipo AXI, aquí se conecta el maestro el cual utiliza el protocolo elegido en SI Protocol.
- **Aclk:** Señal de reloj para el core.
- **Aresetn:** Señal de reset del core, con lógica inversa.

## 7.2 Salidas

 **M\_AXI:** Señal maestra tipo AXI , aquí se conecta el esclavo que utiliza el protocolo elegido en el MI Protocol.

En este proyecto, el AXI Protocol Converter, se utiliza para conectar el core AXI Interconnect, que realiza las transiciones entre los periféricos y el procesador, y el mismo procesador:

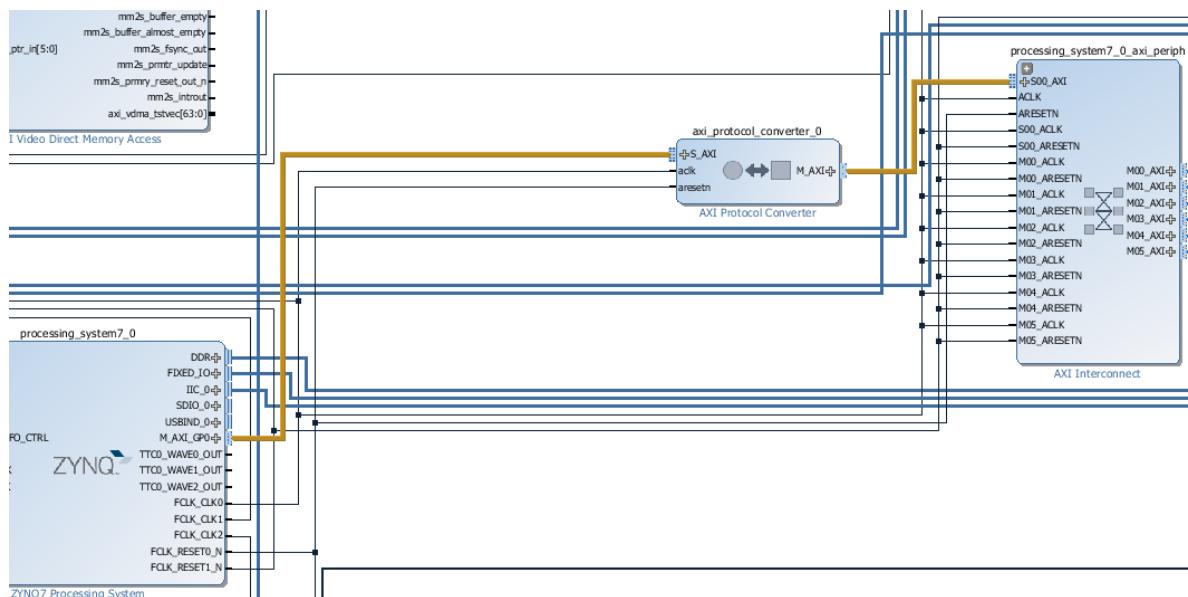


Ilustración 84. Conexión procesador - AXI Interconnect

## 8 Core Utility Vector Logic

Este core permite realizar operaciones de tipo lógico (AND, OR, XOR o NOT) para señales binarias, pudiendo elegir el número de bits que utilizará. En el diseño aquí presentado se ha utilizado la versión v2.0 del core y se ha usado para obtener una señal de reset negada para aquellos cores que precisaban de señal de reset con lógica inversa.

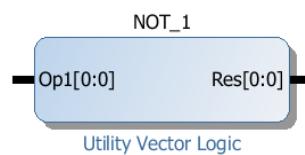


Ilustración 85. Core Utility Vector Logic

Haciendo un clic doble sobre el core, se puede configurar qué operación lógica realizará y el número de bits utilizados:

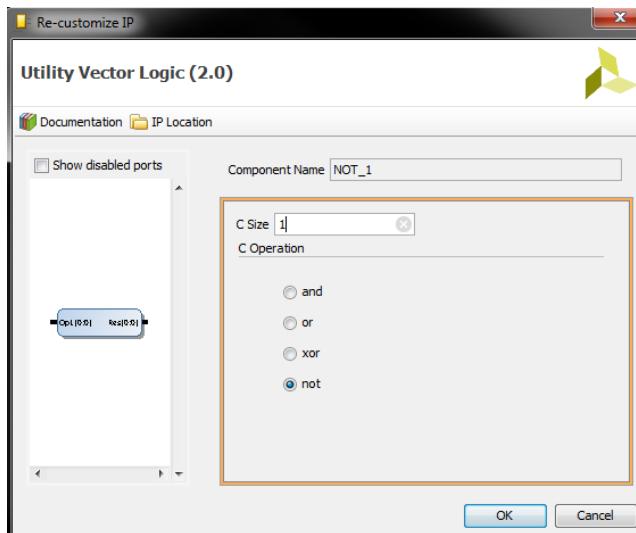


Ilustración 86. Configuración del core Utility Vector Logic

### 8.1 Entradas

**OP:** En este core, la única entrada disponible es el “operando”, es decir, el valor al que se le quiere aplicar la operación lógica seleccionada.

### 8.2 Salidas

**RES:** Será el resultado de aplicar la operación lógica a la entrada “OP”.

## 9 Core AXI GPIO

[XIL,15,F] A través de este core (AXI General Purpose Input/Output) se consigue una interfaz AXI de entrada y salida de propósito general, pudiendo obtener las entradas/salidas propias de la placa, entre otros, como son: LEDs, switches y botones.

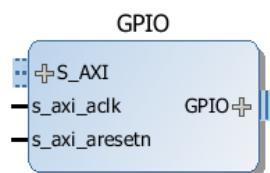


Ilustración 87. Core AXI GPIO

Para este proyecto, se utiliza la versión v2.0 del core que permite utilizar interfaces de tipo AXI4-Lite y puede configurarse para canales GPIO simples o duales (escritura y/o lectura), canales entre 1 y 32 bits, programación dinámica de cada bit del GPIO como entrada o salida, valores de reset individuales para cada bit del registro.

La configuración del core, dependerá de la placa de desarrollo que se esté programando y sus especificaciones.

### 9.1 Entradas:

- **S\_AXI:** Interfaz esclava tipo AXI que se conecta al AXI Interconnect para poder interactuar con el procesador.
- **S\_axi\_aclk:** Señal de reloj para el core.
- **S\_axi\_aresetn:** Señal de reset para el core, con lógica inversa.

### 9.2 Salidas:

- **GPIO:** Será el dispositivo interno de la placa que se quiere controlar, en el caso de este proyecto, se utiliza para leer el valor de la posición actual de los switches de la misma placa de desarrollo.

Estos switches indican su valor con una palabra de 4 bits, siendo el switch más a la izquierda de la placa, el bit más significativo

<i>Posición del Switch</i>	<i>Valor obtenido</i>
	0b0000 = 0
	0b1000 = 8
	0b0101 = 5

El valor de lectura de los LEDs de la placa, o de la pulsación de los botones, se realiza de una forma similar.

Para esto, el GPIO debe convertirse a una señal externa, pinchando sobre la salida con el botón derecho y eligiendo la opción “Make external”:

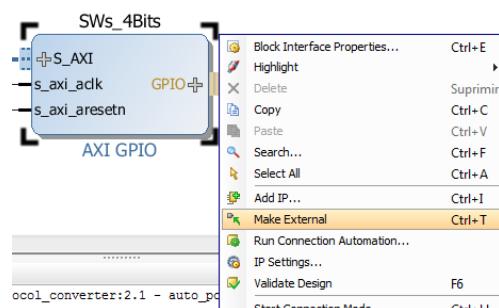


Ilustración 88. Crear señal externa en AXI GPIO

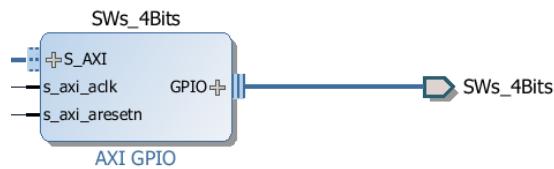


Ilustración 89. Señal externa creada

Y para terminar, debe añadirse al fichero de restricciones para elegir los pines donde esté el dispositivo a controlar:

```

##Switches
#IO_L19N_T3_VREF_35
set_property PACKAGE_PIN G15 [get_ports {sws_4bits_tri_i[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sws_4bits_tri_i[0]}]

#IO_L24P_T3_34
set_property PACKAGE_PIN P15 [get_ports {sws_4bits_tri_i[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sws_4bits_tri_i[1]}]

#IO_L4N_T0_34
set_property PACKAGE_PIN W13 [get_ports {sws_4bits_tri_i[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sws_4bits_tri_i[2]}]

#IO_L9P_T1_DQS_34
set_property PACKAGE_PIN T16 [get_ports {sws_4bits_tri_i[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sws_4bits_tri_i[3]}]

```

Código 13. Configuración de los switches

## 10 Core ZYNQ7 Processing System

<sup>[XIL,15,G]</sup>Por último, el core más importante del diseño, es el ZYNQ7 Processing System, el procesador de la placa de desarrollo, encargado de configurar el procesador con los parámetros que se requieran para el correcto funcionamiento y las necesidades del proyecto. La versión utilizada para este proyecto es la v5.5.

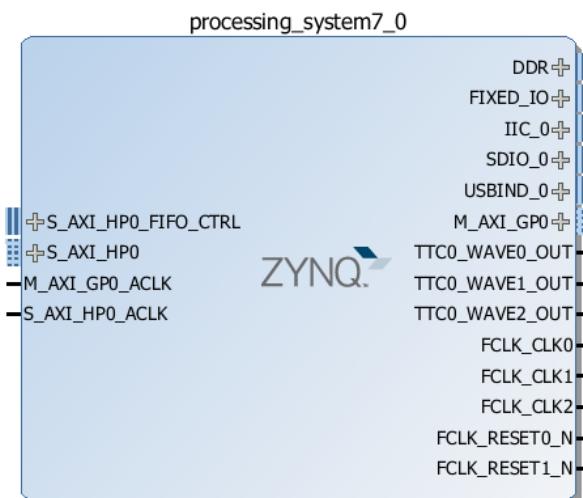


Ilustración 90. Core ZYNQ7 Processing System

El core permite configurar todas las entradas y salidas que sean necesarias para el diseño, así como señales de reloj, o memoria a utilizar. Haciendo doble clic sobre el core:

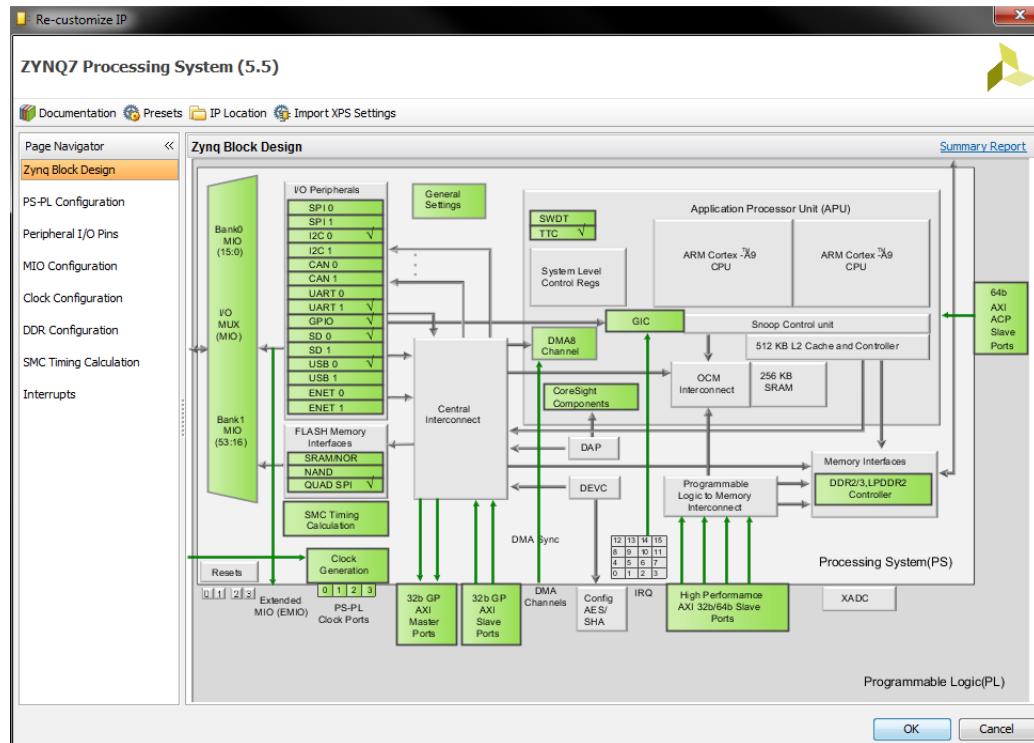


Ilustración 91. Interfaz de configuración del core ZYNQ7 Processing System

En la ilustración anterior se muestran los periféricos configurados para el diseño, marcados con un símbolo “check”. Los cuadros en verde, permiten realizar configuración pulsando sobre ellos y además, se puede navegar por los distintos menús del lateral izquierdo para una configuración más exhaustiva.

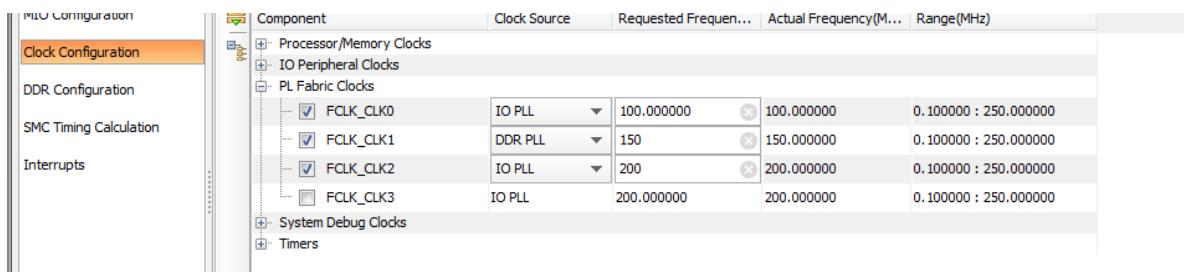
### 10.1 Entradas

- **S\_AXI\_HPO:** Interfaz tipo AXI esclava de altas prestaciones (High Performance), utilizada para las lecturas/escrituras en memoria de manera que se realicen de la forma más rápida posible, para conseguir un rendimiento superior.
- **M\_AXI\_GPO\_ACLK:** Señal de reloj para la interfaz tipo AXI maestra de propósito general (General Purpose) de salida
- **S\_AXI\_HPO\_ACLK:** Señal de reloj para la interfaz tipo AXI esclava de altas prestaciones de entrada.

### 10.2 Salidas

- **DDR:** Señal externa conectada a la memoria DDR de la placa de desarrollo. Generada de forma automática al insertar el core en el diseño.
- **IIC\_0:** Señal externa para protocolo I<sup>2</sup>C.
- **M\_AXI\_GPO:** Interfaz tipo AXI maestra de propósito general (General Purpose) de salida.

-  **FCLK\_CLK0/1/2:** Señales de reloj generadas por el procesador, configuradas a una frecuencia de: 100, 150 y 200 MHz respectivamente, siendo la fuente de los relojes 0 y 2 el IO PLL y la fuente del 1, el DDR PLL.



Component	Clock Source	Requested Frequency(MHz)	Actual Frequency(MHz)	Range(MHz)
Processor/Memory Clocks				
IO Peripheral Clocks				
PL Fabric Clocks				
FCLK_CLK0	IO PLL	100.000000	100.000000	0.100000 : 250.000000
FCLK_CLK1	DDR PLL	150	150.000000	0.100000 : 250.000000
FCLK_CLK2	IO PLL	200	200.000000	0.100000 : 250.000000
FCLK_CLK3	IO PLL	200.000000	200.000000	0.100000 : 250.000000
System Debug Clocks				
Timers				

Ilustración 92. Configuración de los relojes del ZYNQ7 PRocessing System

-  **FCLK\_RESET0/1\_N:** Señales de reset del procesador.